Leviton Absolute Optical Encoder Development 2004

A Major Qualifying Project report:
Submitted to the Faculty of
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements for the
Degree of Bachelor of Science
By

Mudassar Ali Muhammad
(mudassar@wpi.edu) ________________________________

Alex Sanville
(sanville@wpi.edu) ________________________________

Joe Marcin
(marcij@wpi.edu) ________________________________

Date: October 13, 2004

GSFC Mentor:
Doug Leviton
(doug.leviton@gsfc.nasa.gov)
Phone: 301-286-3670
Fax: 301-286-0204

WPI Advisor:
Professor Stephen J. Bitar
(sjbitar@ece.wpi.edu) ____________________________

WPI Advisor:
Professor Freed J. Looft
(fjlooft@ece.wpi.edu) ____________________________

Worcester Polytechnic Institute
100 Institute Road Worcester, MA 01609

Abstract

The Leviton absolute optical encoder is an ultra-precise position-measuring device, capable of detecting linear and angular displacements as small as 1 nm and 0.006 arc seconds, respectively. We developed the first encoder design that produces position updates using the National Semiconductor LM9637 CMOS active pixel sensor, a FPGA, and a DSP. We also proved that just a FPGA could be used with the sensor to achieve even faster results, thus, reducing the overall cost and complexity of the Leviton Encoder.

Acknowledgements

We would like to acknowledge following people for their support throughout our project.

- NASA/GSFC
- Our Mentor Doug Leviton
- Professor Looft & Professor Bitar
- Professor Blandino & Professor Finkel
- Steve Kraft, Fil Parong, David Pfenning
- Salman Sheikh, David Fisher, Ameen Syed
- Scott Smith & Brad Frey
- CS Group & VHDL Group
- Shack Junkies Softball team

Table of Contents

List of Tables

List of Figures

List of Acronyms

- CCD: Charge Coupled Device
- CMOS: Complementary Metal Oxide Semiconductor
- DSK: Development Starter Kit
- DSP: Digital Signal Processor
- EMIF: External Memory Interface
- FIFO: First In First Out
- FPGA: Field Programmable Gate Array
- GSFC: Goddard Space Flight Center
- HST: Hubble Space Telescope
- IC: Integrated Circuit
- LED: Light Emitting Diode
- McBSP: Multi-channel Buffered Serial Port
- MQP: Major Qualifying Project
- NASA: National Aeronautical Space Administration
- PCB: Printed Circuit Board
- RAM: Random Access Memory
- WPI: Worcester Polytechnic Institute

Executive Summary

The Leviton Absolute Encoder

The Leviton absolute optical encoder is an ultra-precise position measuring device. It is capable of detecting linear and angular displacements as small as 1 nm and 0.006 arc seconds, respectively. The encoder achieves its remarkable performance through the use of an active pixel sensing device which captures an image of a backlit linear or rotary scale containing an absolute code pattern (Fowler, 2001).

Applications

The Leviton Encoder has been used for ground support and calibration for numerous NASA missions including the Hubble Space telescope (HST), the James Webb Space Telescope (JWST), Earth Observing System (EOS), and Sub-Millimeter Probe for the Evolution of Cosmic Structures (SPECS) (Jeager, 2002). The encoder was used for astronomical positioning and calibration of sensors that are put onboard satellites. The encoder allows a satellite to calibrate its sensors faster and more precisely.

Although the Encoder has only been used for NASA science missions, there is a wide range of applications for the Leviton Encoder technology in the commercial world. One major application for the encoder is making optical beam steering extremely precise. This allows for improved laser-cutting of metals and glass, accurate output redirection of one optical fiber into another, and improved military applications such as laser guided weaponry along with many others (Leviton, Personal Communications, 09/02/2004).

Problem Statement

Originally the encoder was implemented with a CCD (Charge Coupled Device) sensor to capture the image of a backlit scale, which also required an analog front-end. The 2003 Team replaced the CCD sensor with a CMOS (Complementary Metal Oxide Semiconductor) Active Pixel sensor, which outputs digital image data, hence, removing the need for the analog front-end. Replacing the CCD sensor also increased the maximum speed of the image data available to the DSP, from 12MHz compare to the 27MHz pixel data rate of the CMOS Active Pixel sensor. Although, the sensor is sending the data at a much faster rate of 27MHz, the DSP is only capable of receiving data at 8MHz. Due to this speed difference, a bottleneck occurs between the sensor and the DSP, resulting in data overrun. Consequently, the DSP cannot perform the image processing algorithm correctly.

Image Processing Algorithm

To fully understand the reason behind the implementation of our design, it is necessary to have a full understanding of the image-processing algorithm. A detailed explanation of the image-processing algorithm can be found in Appendix A. For our purposes, the actual calculation of the position is the same. However, the technique we use to navigate through the image is different than the one described in Appendix A. Our image-processing algorithm begins navigating through the image by first locating the pixel columns which contain the vertical lines. These columns are determined by summing, along each pixel column, from the top to the bottom of each vertical line. The summation results in a one-dimensional array, from which the vertical lines are located by comparing each element in the array to a threshold value. The threshold value is determined to be half of the maximum value of a summed column. Then the centroid is calculated, as described in Appendix A. After the centroid for each visible vertical line is known, the code bits are located through a known pixel offset for that particular scale. Once the code bits are interpreted, the position is calculated as described in Appendix A. The row markers are unused in this fashion, which allows for a simpler implementation.

Implemented Solution

We discovered that a relatively simple FPGA could perform all the summing functions at the speeds we required. The pixel data of each column coming out of the sensor, at 25MHz—the clock speed provided to the sensor—was accumulated. A threshold value was set to determine the vertical lines. Once the number of pixels that contain the vertical lines was determined by comparing the summed value to the threshold vale, the FPGA was set to output this accumulated data to the DSP. Then the accumulator was reset and the FPGA summed the part of the column that contains the first row of code bits. After the first row of code bits was determined through the same process as the vertical lines, the accumulator was reset for the second row of code bits.

Although the FPGA part was fully functional, the DSP still did not take in the accumulated data at the required speed. We then stored the accumulated values into a RAM of the FPGA. With increased familiarity with the FPGA, we recognized that the whole image-processing algorithm can be performed on the FPGA. Therefore, we changed our focus to designing logic for the image-processing algorithm. After completing most of the logic for the algorithm, we discovered that 32-bit floating point arithmetic library in VHDL—required for computing the position updates—will require another ten weeks, thus making it impossible for us to use just an FPGA for the whole image-processing algorithm.

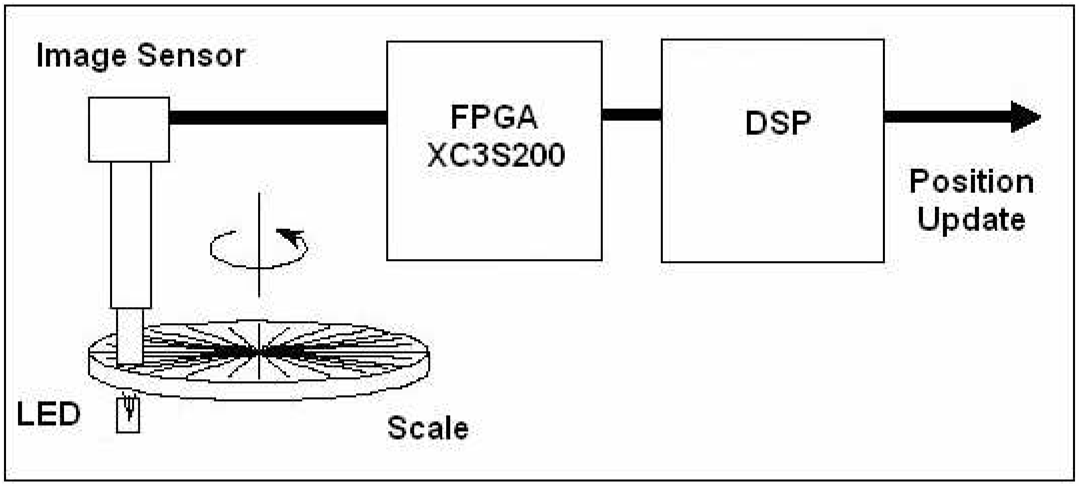
On the other hand, the DSP can be easily used to perform 32-bit floating point arithmetic. At this point, we had the FPGA logic output three operands (vertical lines, first code bits, and second code bits), required for the image-processing algorithm, into an FPGA RAM. As explained in our Results chapter, the block data transfer to the DSP is much faster than sending values one at time. Therefore, we performed a block transfer of the FPGA RAM to the DSP over the EMIF. After each transfer and with simple C code, we obtained the position updates.

Ideal Solution

The ideal solution would be to have the FPGA perform the whole image-processing algorithm. This would mean that the encoder will only consist of a sensor and an FPGA, as two main components. Therefore, removing DSP from the encoder, this will significantly reduce the overall cost and complexity of the device. We have proved that the FPGA is capable of performing the whole algorithm, provided that a working 32-bit floating point VHDL library is available. It must be noted that there are such libraries available that will perform floating point operations, however, the lack of time restricted us from getting those libraries to work accordingly with our project.

Encoder Design

The Leviton Absolute Encoder now relies on FPGA to perform most of the image-processing algorithm. The following is a high level block diagram of the new Leviton Encoder:



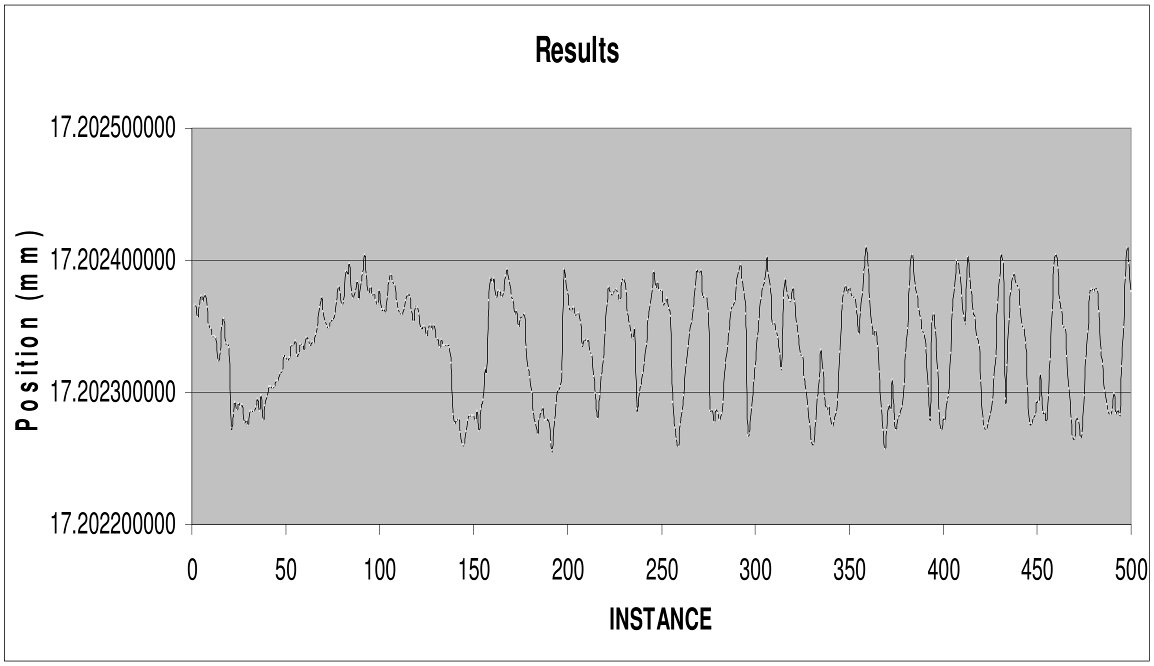
Ideally, the encoder will only have the Image sensor and the FPGA. However, in the above diagram, the DSP is utilized to obtain position updates. Also, DSP provides various options on collecting position updates and presenting it in a data file, which can later be used in a spread sheet program for graphical analysis.

Position Updates

Two of the results we obtained were under the same conditions. One was the encoder just placed on the table and the second, in which the encoder scale was moved eighteen times with about a 50 micron increment every time. The condition in which these two results were obtained were the worst case conditions, in terms of window size. Following are the conditions:

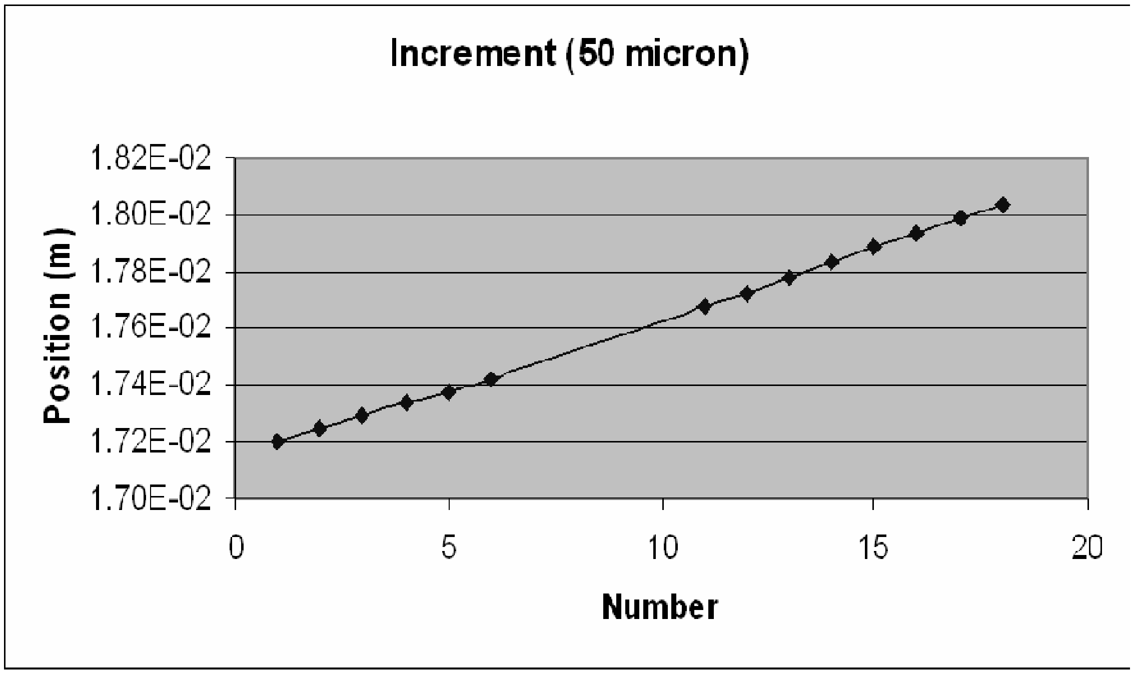
- Sensor Master Clock = 25MHz
- Window = 640 x 480
- Row Delay = 8h
- No lens

The frame rate under these conditions is 63.2 frames/second. The Following figure shows 500 position updates obtained with the encoder just placed on a table.



500 Position Updates

The previous chart shows the 500 position values on a 100nm division. Since the setup was not isolated from vibration and noise of other equipment on the table, the resulting position is oscillating. The next figure shows the position updates as the encoder was moved vertically on the sensor 18 times by about 50 microns.



Increments

It must be noted that each increment was approximately 50 microns. The four data values, number seven through eleven, not shown in the above graph were excluded because of their inaccuracy, solely due to the setup of the test. The next eight values on the graph prove that, in fact, the setup of the encoder scale on the sensor caused inaccurate results in those four increments. To further prove that the setup was to be blamed for skewed values, the VGA output was obtained which revealed that the code bits were cutoff, thus, the sensor was sending incomplete pixel data, in terms of the encoder scale.

Recommendations

The following are the recommendations to further improve the encoder design and also to make it more marketable.

- Use FPGA to fully implement the image-processing algorithm
- Incorporate all window sizes and scale patterns
- Separate FPGA and the DSP for a PCB design
- Initialize the sensor using FPGA as described by 2003 team
- Create a VGA output for the encoder

1.0 Introduction

The Leviton Encoder technology is in its eighth year of development at the Goddard Space Flight Center (GSFC) of the National Aeronautics and Space Administration (NASA), located in Greenbelt, Maryland. Douglas B. Leviton is the inventor of this encoding technique. This chapter will briefly describe the Leviton Encoder and its applications along with the problem statement for this project.

1.1 The Leviton Absolute Encoder

The Leviton absolute optical encoder is an ultra-precise position measuring device. It is capable of detecting linear and angular displacements as small as 1 nm and 0.006 arc seconds, respectively. The encoder achieves its remarkable performance through the use of an active pixel sensing device which captures an image of a backlit linear or rotary scale containing an absolute code pattern (Fowler, 2001).

Linear and rotary scales are in many ways similar to a ruler and a protractor, respectively. Both scales contain unique markings corresponding to their position in the applicable field of motion, which allows the device to determine its exact absolute position even after losing power or upon restart.

1.2 Problem Statement

Originally the encoder used a CCD (Charge Coupled Device) sensor to capture the image of a backlit scale, which also required an analog front-end. The 2003 Team replaced the CCD sensor with a CMOS (Complementary Metal Oxide Semiconductor) Active Pixel sensor, which outputs digital image data, hence, removing the need for the analog front-end. Replacing the CCD sensor also increased the speed of the image data available to the DSP because the active pixel sensor produces digital image data at 27MHz, more than twice as fast as the CCD sensor (12MHz).

Although, the sensor is sending the data at a much faster rate of 27MHz, the DSP is only capable of receiving data at 8MHz. Due to this speed difference, a bottleneck occurs between the sensor and the DSP, resulting in data overrun. Consequently, the DSP cannot perform the image processing algorithm correctly. Project goal and tasks set to complete this project are discussed in chapter 3.

1.3 Applications

The Leviton Encoder has been used for ground support and calibration for numerous NASA missions including the Hubble Space telescope (HST), the James Webb Space Telescope (JWST), Earth Observing System (EOS), and Sub-Millimeter Probe for the Evolution of Cosmic Structures (SPECS) (Jeager, 2002). The encoder was used for astronomical positioning and calibration of sensors that are put onboard satellites. The encoder allows satellite to calibrate its sensors faster and more precisely than before.

Although the Encoder has only been used for NASA science missions, there is a wide range of applications for the Leviton Encoder technology in the commercial world. One major application for the encoder is making optical beam steering extremely precise. This will allow for improved laser-cutting of metals and glass, accurate output redirection of one optical fiber into another, and improved military applications such as laser guided weaponry along with many others (Leviton, Personal Communications, 09/02/2004).

One more application of the Leviton Encoder would be for wafer steppers used in integrated circuit (IC) production. The Encoder can be used to enhance the accuracy and precision of placing extremely small components in the silicon wafer. Increased accuracy and precision will make integrated circuit production more efficient by reducing the number of faulty chips (Fowler, 2001).

1.4 Summary

The Leviton Encoder has the potential to become a very useful positioning measuring device in many fields. Upon completion of this project, this encoder will ideally become a standalone device and be entirely packaged in a 2" cube, thus providing NASA with an excellent tool for positioning in various space-related and earthly applications along with potentially aiding the commercial world of motion control. The next chapter will provide extensive background information about of this project.

2.0 Background

This chapter begins by providing background information on NASA and its Goddard Space Flight Center, along with our mentor Doug B. Leviton who is an optical physicist at the GSFC. In addition, the concept of encoders, in general, and the Leviton Absolute Optical Encoder, in particular, is discussed in detail. This chapter also entails a brief summary of the work that was completed by previous WPI project groups in the development of Leviton Absolute Optical Encoder. Moreover, some of the key hardware components and their respective functions in the encoder are also explained in detail.

2.1 National Aeronautics and Space Administration

NASA was formed on October 1, 1958 by the United States government under the following motto:

> "An Act to provide for research into the problems of flight within and outside the Earth's atmosphere, and for other purposes" (Garber, Launius, 2002)

NASA is responsible of human space flight, aeronautics, space science, and space applications. Since its creation, NASA has made many historical achievements, among which are landing on the moon and the Hubble Space Telescope.

2.1.1 Goddard Space Flight Center

The Goddard Space Flight Center is located in Greenbelt, Maryland. The GSFC is named after rocket propulsion pioneer Robert Goddard, who attended Worcester Polytechnic Institute. The GSFC has the following vision statement (GSFC, 1998):

> "We revolutionize knowledge of the Earth and the universe through scientific discovery from space to enhance life on Earth."

The Goddard team consists of America's premier scientists and engineers who are devoted in research of Space Science, Earth Science, and Technology (GSFC, 1998).

2.1.2 Mentor Information

Douglas B. Leviton, an optical physicist at the GSFC, received his Bachelors' Degree in 1981 from Emory University, Georgia and obtained his Masters' Degree in

Applied Physics from Georgia Institute of Technology. His research focus is mainly in optics. His invention, the Leviton Absolute Optical Encoder, was awarded NASA's Government Invention of the Year in 1999. Mr. Leviton has also worked on numerous NASA science missions including the Hubble Space Telescope, Solar and Heliospheric Observatory, and Earth Observing System (NTB, 2000).

2.2 Encoders

An encoder is a position sensing device that works on either a linear or rotary scale. The scale is similar to a ruler with markings indicating the position. The encoder reads information from this scale to determine its location. There are two types of encoders incremental and absolute. An incremental encoder uses a scale with incremental markings. This allows the incremental encoder to determine its position relative to a certain location. An absolute encoder has unique markings on its scale that allows it to determine its exact location in the applicable field. This allows an absolute encoder to always know where it is on a particular scale. On the other hand, if an incremental encoder was to lose power and upon restarting the encoder would no longer know its position in the applicable field (Gieras, 2000).

2.3 Development History of the Leviton Absolute Encoder

Prior to WPI's involvement in the Leviton Encoder, the encoder consisted primarily of a PC controlling a CCD (charge coupled device) sensor and then running the image processing algorithm. This design only allowed for speeds of 10Hz–due to the limits of the PC data interface and relatively slow computing speeds. Since this design required a PC to run a single image processing algorithm, the size of the encoder was not ideal.

For the past three years, MQP teams from WPI have worked to enhance the performance of the encoder and make it marketable. The following table highlights the major achievements of previous three years teams (Fowler, 2001).

Team	Encoder Size	Data Coordination	Position Updates (Hz)	Sensor	Major Contribution
Original	PC	PC	10	CCD	-
2001	5’x8’x1’	FPGA	1529.2	CCD	Introduced DSP
2002	5’ x 4’	FIFO	6000 Theoretical	CCD	Separated PC
2003	Not completed	FIFO	Not completed	Active pixel Sensor	Introduced Active Pixel Sensor

2.1 Major Achievements by the Previous MQP Groups

The next section provides a summary of the work of previous three groups.

2.3.1 The 2001 Team

The 2001 WPI project team began to free the encoder from the PC and increase the speed of the device. They introduced a DSP for performing image processing algorithm, previously run by a PC. They used a Texas Instruments TMS320C6711 DSP. Although they were able to get the DSP to perform the image processing algorithm, they still required a PC to save the algorithm code.

They were using a CCD sensor, which meant that they had to first develop an analog front-end for converting analog data into a digital form for the DSP to read. The 2001 project team also used a FPGA (field programmable gate array) to coordinate data transfers between the DSP and the CCD sensor. This new design provided 1259.2 position updates per second (Fowler, 2001).

2.3.2 The 2002 Team

The 2002 WPI project team was able to completely remove the PC from the encoder by storing the image processing algorithm on a flash memory. This allowed for a stand alone device that would have a wider range of applications. They also introduced a FIFO (first-in first-out) memory block to the encoder for packing the data before it was sent to the DSP. By packing data, transfer rate bottlenecks were reduced and it allowed for theoretical position update speeds greater than 6 KHz (Harvey et al., 2003). Also, the use of a flash memory for storing the image processing algorithm code reduced the size of the Leviton Encoder. The encoder could now fit onto a single 5"x 4" PCB (Printed circuit board) (Jaeger et al., 2002).

2.3.3 The 2003 Team

The 2003 WPI team focused on the limitations of the current sensor and began implementing a new design using a National Semiconductors LM9637 CMOS active pixel sensor. This sensor outputs digital data so there was no need for an analog front end. This sensor was also less expensive and used much less power than the previous CCD device. The CMOS sensor also has several features which provide greater flexibility to the design.

The 2003 team was successful in initializing the sensor and developed a start up routine for it. Problems arose when transferring data from the sensor to the DSP for image processing. First, the sensor was connected directly to the DSP and used the pixel clock for interfacing with the DSP. Theoretically, it should have worked properly. However, in the implementation the digital image data was being lost because the DSP was not taking the data in at 12MHz, the minimum speed of the sensor. One attempt involved using a FIFO to buffer data so that the DSP could get data when it wanted rather than having it accept data at given intervals. This method worked better but could not achieve the minimum speed of 12MHz required by the sensor (Harvey et al., 2003). The picture of the final design is shown below in figure (2.1).

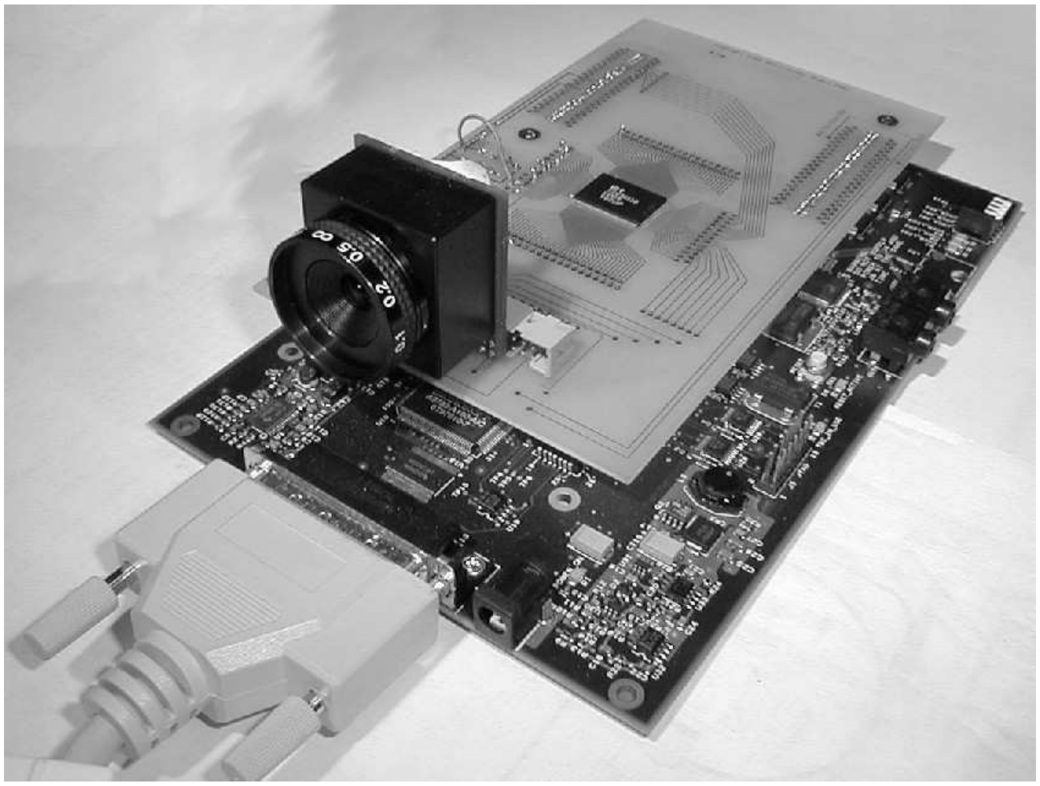


Figure 1:Final Product of the 2003 MQP Group (Harvey, 2003)

2.4 Main Components and Functionality

Currently, the Leviton Absolute Encoder uses a back-lit encoder scale, a CMOS active pixel sensor, FPGA and a DSP. This section explains these components in detail and how they are used in the overall encoder design.

2.4.1 The Scale and Image-Processing Algorithm

The scale used for the Leviton Encoder is vital to the encoder. The scale is the part that the sensor reads to determine the exact position of the object. The scale consists primarily of vertical lines and code bits. The vertical lines are like the markings on a ruler and the code bits are like the numbers. Moving along the scale, the sensor captures an image of the scale and then sends this image data to the DSP. The DSP then runs the image-processing algorithm on this data. The image-processing algorithm detects the vertical lines. When the algorithm locates a vertical line, it then looks for the code bits, which are unique to each position on the scale.

The scales that we will be working with come in two forms, the “classical image” or the “binnable image”. The classical image has code bits located at the bottom of each line. The binnable image contains only vertical information where the code bits are located between the lines. Figure 2.2 (Jaeger, et al., 2002) illustrates the differences between the classical image and the binnable image.

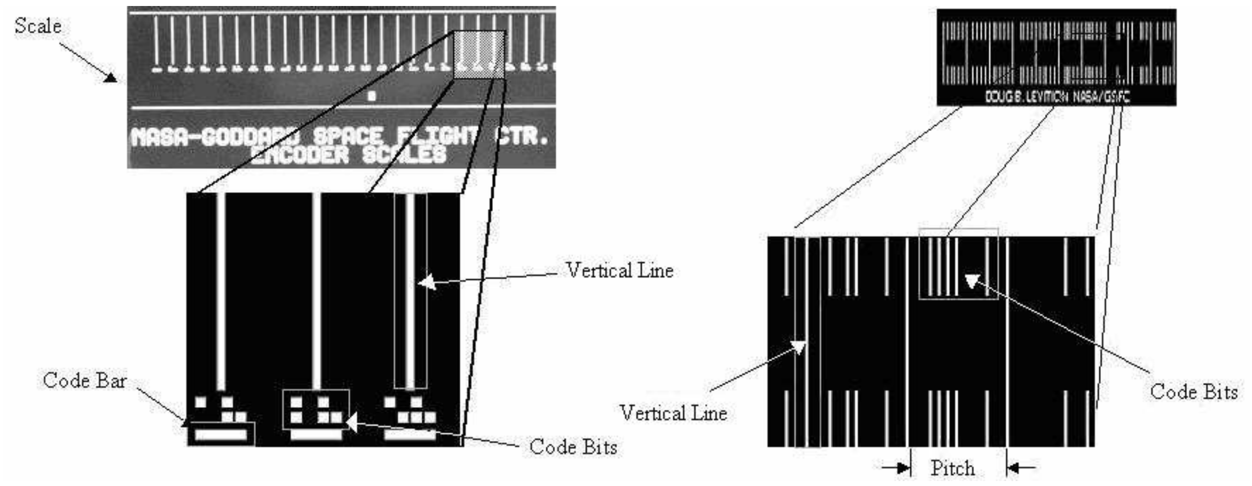


Figure 2:Classical Image (left) and Binnable Image (right) (Jaeger, 2002)

Each type of image has its advantages and disadvantages. The classical image takes longer for the algorithm to interpret but is more accurate than the binnable image due to its code bits. The binnable image can run through the algorithm much quicker but with only vertical information it has a higher noise to signal ratio, which reduces accuracy (Harvey et al., 2003).

The image-processing algorithm developed for reading these scales, gives the Leviton Absolute Encoder its nanometer resolution. The algorithm varies slightly for the “classical image” and the vertically “binnable image”. Essentially, for each image, the image-processing algorithm goes through a process of finding all of the vertical lines in the image, determining which lines are in the image (interpreting code bits), and then calculates the offset of each line compared to the center of the image. An in depth explanation of the image processing algorithm can be found in Appendix A.

2.4.2 The Sensor

The sensor currently used on the Leviton Encoder is National Semiconductors LM9637 CMOS active pixel sensor. This sensor was chosen by the 2003 team due to its many configuration options, digital output, low cost, and its low power consumption compared to other sensors. A block diagram of the sensor is shown below in figure 2.3

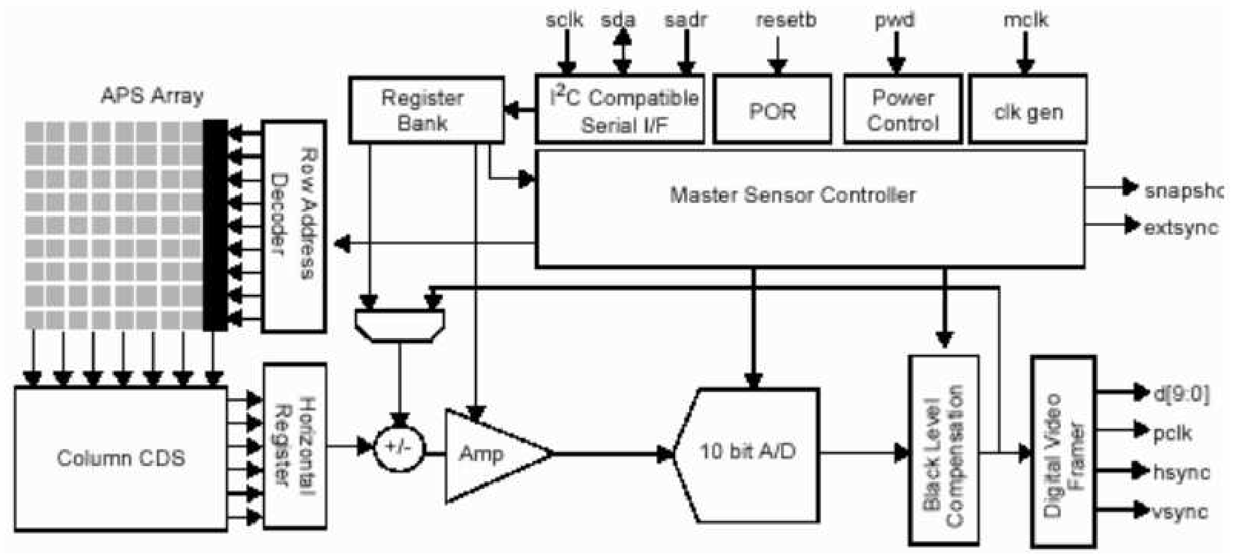


Figure 3:Sensor LM9637 Block Diagram (National, 2002a)

The active pixel sensor is controlled by a two line serial input more commonly known as I^2C (Inter-Integrated Circuit Control). Upon start up there are many registers that must be programmed on the LM9637 to get the desired operation. Using the I^2C interface the 2003 project group were successful in initializing the registers and have developed a method for doing so.

This particular sensor is capable of varying window sizes with the largest being 640 x 480 pixels. Smaller windowing sizes can be programmed upon initialization of the registers. Depending on the configuration of the device the sensor will read each pixel out as an 8-bit or a 10-bit digital number. In the standard configuration, the sensor will read out the upper left pixel first and then continue on in a similar fashion to reading a book. Also depending on whether the sensor is run in master or slave mode there will be horizontal and vertical synchronization signals that will become outputs or inputs respectively. This allows for control over the speed at which the sensor steps through the pixels (National Semiconductor, 2002b).

2.4.3 The FPGA

The FPGA used in the Leviton Encoder is the Xilinx Spartan-3 XC3S200. This FPGA was chosen for the flexibility that it can provide to the system. This FPGA has 200,000 logic gates, 173 I/O pins, and 216Kb of internal RAM. The FPGA's main task is to collect the pixel data coming from the sensor, accumulate the data, and then store the data for the DSP to access later. The Figure 2.4 shows an overview of the development board for the FPGA.

The Development board came with many useful features. It has an on board flash memory (2) which provides an easy way to make designs nonvolatile. It also has three 40 pin expansion connectors (19-21) which allows direct connection of signals to the I/O pins of the FPGA. This board also contains several buttons (13), switches (11), LED's (12), and four seven segment displays (10). The buttons and lights take up some of the I/O pins of the FPGA but are extremely helpful during the prototyping process. The following figure shows the top view of the FPGA development board.

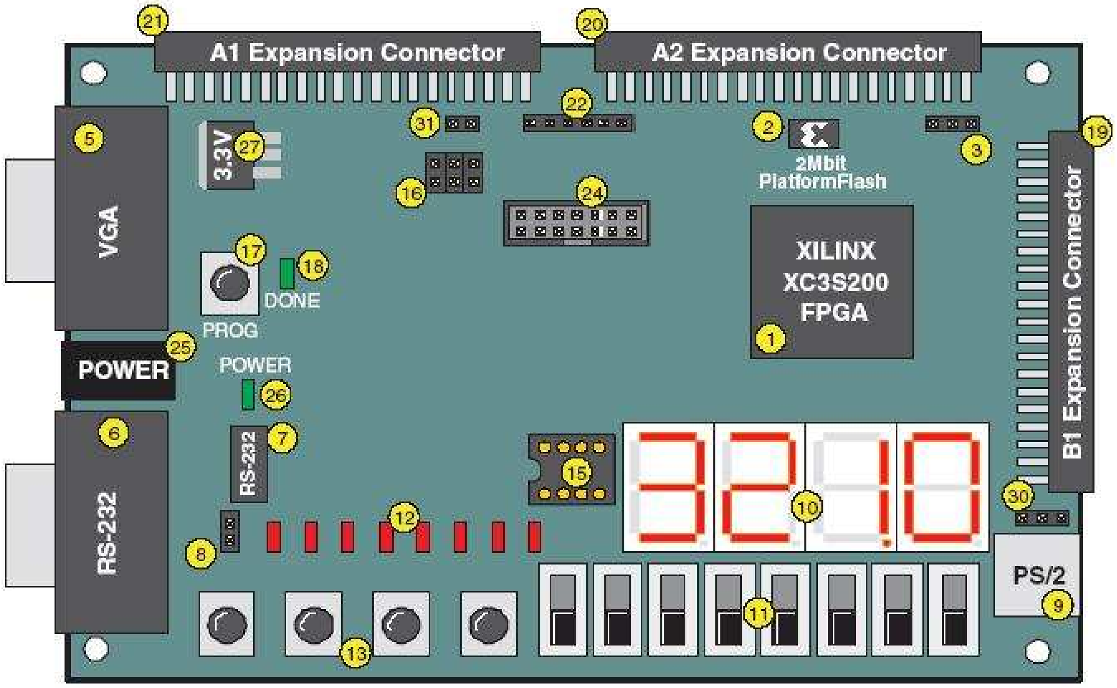


Figure 4 Spartan-3 Development Board Lay Out (Xilinx, 2004b)

2.4.4 The DSP and its Functions

The DSP currently being used is the Texas Instruments TMS320C6711. The primary functions of the DSP is to initialize the CMOS pixel sensor and perform the image-processing algorithm with all the operand for the calculations given by the FPGA, and then output the result. The block diagram for the TMS320C6711x family of processors is shown below in figure 2.5(Texas Instrument, 2004c).

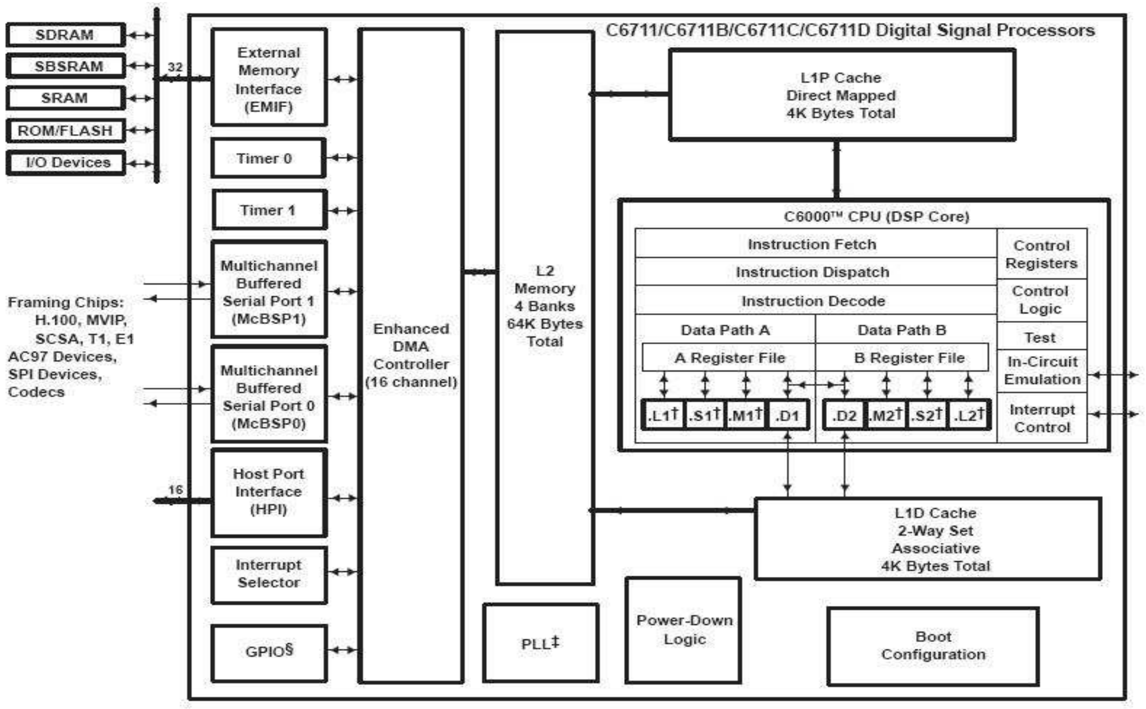

C6711/C6711B/C6711C/C6711D Digital Signal Processors
SDRAM
SBSRAM
SRAM
ROM/FLASH
I/O Devices
32
External Memory Interface (EMIF)
Timer 0
Timer 1
Framing Chips: H.100, MVIP, SCSA, T1, E1
AC97 Devices, SPI Devices, Codecs
Multichannel Buffered Serial Port 1 (McBSP1)
Multichannel Buffered Serial Port 0 (McBSP0)
16
Host Port Interface (HPI)
Interrupt Selector
GPIO§
Enhanced DMA Controller (16 channel)
L2 Memory 4 Banks 64K Bytes Total
L1P Cache Direct Mapped 4K Bytes Total
C6000™ CPU (DSP Core)
Instruction Fetch
Instruction Dispatch
Instruction Decode
Data Path A
Data Path B
A Register File
B Register File
.L1† .S1† .M1† .D1
.D2 .M2† .S2† .L2†
Control Registers
Control Logic
Test
In-Circuit Emulation
Interrupt Control
L1D Cache 2-Way Set Associative 4K Bytes Total
PLL‡
Power-Down Logic
Boot Configuration

Figure 5 Block Diagram of the TMS320C6711x family (Texas, 2004c)

For a detailed explanation of these features see Appendix B.

2.5 Summary

This chapter has covered some of the major background information required to understand the Leviton Absolute Optical Encoder, in context to this project. It also provides a summary on the development of the encoder over the years. The next chapter will describe the goal of this project and various tasks that are required for this project to be successfully completed.

3.0 Problem Statement

This chapter describes the problem being addressed by this MQP and states what needs to be accomplished for a successful project. The following section details the main problem that is being addressed. The rest of the chapter details the main goal and objectives of this project along with the tasks that must be completed to achieve the project goal. The tasks are divided into primary and secondary tasks. Primary tasks must be completed in order for a successful MQP. Secondary tasks are not required to solve the problem, but with resources permitting, can be done to improve the usefulness and marketability of the Leviton Absolute Encoder.

3.1 Project Problem

In the ideal system layout of the Leviton encoder, the sensor would send 10-bit of digital image data to the DSP at 27MHz. Unfortunately, the DSP is only capable of receiving data at 8MHz. Due to the speed difference, a bottleneck occurs between the sensor and the DSP, resulting in data overrun. Consequently, the DSP does not get the appropriate data to perform the image-processing algorithm.

3.2 Project Goal

The goal of this project is to design and implement a solution that will store the data coming out of the sensor at 27MHz and then transfer the data to the DSP at a maximum speed of 8MHz, without any data overrun. This solution must also allow the DSP to perform the image processing algorithm and send out at least 1200 position updates per second.

3.3 Objectives

We set the following objectives to achieve our project goal:

- Separate the sensor from the evaluation kit
- Initialize the sensor using the DSP
- Achieve a rate of 1200 position updates per second
- All scale patterns and window sizes
- Run the sensor in Master (free-running) mode without data overrun
- Incorporate the encoder design in a 2-inch cube

- Keep production costs under $200

3.4 Primary task

To achieve our project goal, the following six tasks needed to be completed:

- Research the sensor to determine its capabilities and setup.
- Research the DSP to determine its capabilities and setup.
- Fully understand the image processing algorithm.
- Come up with feasible solutions for our problem and evaluate each solution
- Pick the best solution and test it, if successful implement it.

3.5 Secondary Tasks

The following goals were set as secondary goals to increase to the quality of the Leviton Absolute Optical Encoder:

- Add more functionality to our design to accommodate different scale types and window sizes.
- Implement the design on a stack of PCB's which can fit into a two-inch cube.
- Research other DSP options that would improve upon the current design.
- Work on an image capturing device for alignment purposes.
- Research what needs to be done to current design in order for it to read a Cartesian scale.

3.6 Detailed Task Descriptions

This section will go into more detail on why the primary tasks needed to be completed for this project.

3.6.1 Task 1

- Research the sensor to determine its capabilities and setup.

The device that captures the image and reads out the digital image data to the DSP is the National Semiconductor's LM9637 CMOS Active Pixel Sensor. We needed to understand how the sensor is initialized and its features. This also involved learning Snaps and Visual Basic programs.

3.6.2 Task 2

- Research the DSP to determine its capabilities and setup.

We needed to educate ourselves on the current DSP and its functions. We needed to familiarize ourselves with all the feature of the DSP, including any clocks, data ports, control options, and available memory. There were also other factors that we had to take into account such as power consumption, price, size, and complexity.

3.6.3 Task 3

- Fully understand the image processing algorithm.

We needed to understand how the encoder achieved its remarkable accuracy and precisions. This involved learning how the data coming out of the sensor is computed and the mathematical reasoning behind it.

3.6.4 Task 4

- Come up with feasible solutions for our problem and evaluate each solution

Our solution had to be within a reasonable price range and not be overly complicated so that we can implement our design within our project timeline. We researched numerous devices that would solve our problem and consulted with our mentor. This was on of the most time consuming aspect of the project and involved many ours of brainstorming and evaluating datasheets.

3.6.5 Task 5

- Pick the best solution and test it, if successful implement it.

Once we decided on our solution, we consulted our mentor. After the mentor's approval, we implemented our solution. This required testing and verifying at every stage. When our solution became functional, we presented it to our mentor.

3.7 Summary

This section has defined the project problem and the goal of this project. It also states, in detail, the tasks that were required to complete this project successfully. The next section describes how the tasks were completed and along with covering the various implementations of ideas.

4.0 Methodology

This chapter discusses the methodology used and the path taken to complete this project. First, it describes the research that was needed to understand the project and how that research was applied to form a solution for the project problem. Later, this chapter explains the experimentation phase of the project by chronologically describing the steps taken to achieve the project goal. It concludes by describing the final implementation of our solution for the Leviton Encoder.

4.1 Research

The research phase of this project began in PQP several months before we started this project at NASA/GSFC. The main purpose of the research was to become familiar with the encoding concept and the major components of the device, such as the DSP, to determine if a new DSP should be bought to replace the old one. We first began researching the previous three years of MQP reports on the Leviton Encoder to fully understand the development that the encoder has gone through over the past three years. After arriving at NASA, we focused our research on the sensor and the DSP, simultaneously, to fully understand their functionality and how we can take advantage of their capabilities.

For the sensor, we followed the procedure that the 2003 team used to interface with the sensor. We first familiarized ourselves with the SNAPS EVAL program that came with the Sensor Evaluation Kit. Then we began to learn all the registers that can be programmed to obtain the desired settings for running the sensor. After gaining good understanding of the sensors registers, we used the Visual Basic program developed by the 2003 team to program the sensor using a digital I/O card. Using the Visual Basic program to initialize the sensors registers provided us freedom from the evaluation kit and confirmed that we were programming the registers in the correctly.

For the DSP, we needed to educate ourselves with the Code Composer software from Texas Instruments, which is used to program the DSP. Code Composer comes with many tutorials each on a different function of the software. These tutorials were most helpful in teaching us how to navigate through the Code Composer environment.

However, they were not as helpful in providing information on controlling inputs and outputs, which was crucial to this project.

We investigated last year's code to see how they utilized the inputs and outputs of the DSK. This was extremely helpful in determining how to control the EMIF and other various control signals. Due to the complexity of the DSP, we decided not to limit our research to set amount of time. Throughout the project we constantly learned new functions of the DSP and how we could utilize them to better our project. This proved to be useful for last years project group and allowed them to initialize the CMOS image sensor through the DSP, thus, eliminating unnecessary components (Harvey, 2003).

After determining the project problem, we began to look at various different options of slowing down the image for the DSP, so it can perform the image-processing algorithm. We also studied the image-processing algorithm to see if it could be changed, without compromising the final outcome, to better suit the DSP. Then we recognized that the data coming out of the sensor could be accumulated before it has to go to the DSP, as will be explained in detail in the next chapter.

For accumulating data at the same speed as it comes out of the sensor, we looked into FPGA's. We began our research to find the best FPGA for our solution. We focused on getting a FPGA that had high processing speed with enough logic gates to do this job. When choosing the FPGA, we also considered the cost of the Evaluation kit and the cost of the chip as well.

Once we obtained the FPGA development board we began writing VHDL code to do the accumulation. To do this we had to relearn the ISE software that we had used in school. This was a fairly quick process since all of us were familiar with the software and its interface. There were some major differences however in the actual programming of the device. Unlike the chips we used in school this FPGA had on chip block memory that could be used in a variety of ways. The FPGA also had several other features that such as a clock manager that proved to be useful in the end. Once we had a solid understanding of the FPGA and what we wanted to do we began working on the code for programming the FPGA.

4.2 Experimentation

After completing significant research on the sensor, we began our experimentation phase of the project with the SNAPS program, which came with the Sensor Evaluation Kit. We primarily used the SNAPS program to study the effects of changing sensor registers values by observing the effects of the registers on a VGA computer monitor. This experiment provided us with a detailed understanding of the sensors registers. We then used the method described by the 2003 team, in Appendix C, to learn how to separate the sensor from the evaluation kit.

After separating the sensor from the evaluation kit, we used the program developed by last years MQP for programming the sensor registers, the code is included in the CD. This Visual Basic program used a digital I/O card to interface with the sensor. We again followed 2003's procedure which is included in Appendix C. After successfully programming the registers using the Visual Basic program we moved on to programming the registers using the DSP.

Last years DSP register programming code proved to be well-documented and easy to use. We made a few minor changes to some of the registers according to the requirements of our design. Most notably, the sensor operational mode was changed from Slave mode to Master (free-running) mode, which is a requirement of our design. This process was fairly short due to excellent documentation in last years MQP. The DSP code for the sensor initialization can be found in Appendix D. Last year's MQP documents the sensor initialization procedures in detail and can be found in Appendix C.

Our original solution was to have the FPGA accumulate data and then pass the accumulated data piece by piece to the DSP. Once the DSP had all the accumulated data it would then perform the image-processing algorithm. While developing the code for the FPGA we designed a simple test that would allow us to confirm that we were in fact accumulating data. To do this we simply tied the top eight bits of the 20 bit accumulated data to the LED's on the development board. This essentially creates a light intensity meter. We verified that when we put a flashlight in front of the sensor the top eight bits represented by the LED's increased. We also confirmed that when the sensor was covered the 8-bit value decreased. This proved that the FPGA was capable of accumulating the 10-bit sensor data.

Once we were sure our FPGA logic was working correctly we then added the DSP to the system. We set up the DSP to take each accumulated value as it was generated by the FPGA. However, while testing this approach we discovered that the DSP was not able to handle data transfer in this manner. We came up with two solutions to get around this problem. We could store a whole image of accumulated data on the FPGA and then have the DSP perform a block transfer. A block transfer allows for much higher transfer speeds and eliminates the data transfer problem. The second solution would simply have the FPGA store the accumulated data internally and then perform the image-processing algorithm itself.

We decided to have the FPGA perform the image-processing algorithm. Due to our limited time on the project, there was not enough time left for writing code for the FPGA to handle different types of Encoder scales. However, to prove that the FPGA can in fact perform the image-processing algorithm, we decided to implement logic for only one encoder scale pattern. The details of the scale are provided in the next chapter. To get the algorithm to work in the FPGA, we stored the 20bit accumulated data into a ram created in the FPGA and then found the vertical lines. Once we found the vertical lines it was possible to find the code bits. We began implementing this design but quickly ran into a problem that could not easily be solved in the time we had left. In order to perform the image-processing algorithm it is necessary to do some floating point arithmetic. This is very easy to do in the DSP. On the other hand it is not very easily done in VHDL. The floating point math could be done, but after consulting with our advisors, we found that it would take probably ten weeks or so to implement. This brought us back to using the DSP to performing the image-processing algorithm.

In order to perform a block transfer of accumulated data from the FPGA to the DSP we needed to learn more about the EMIF and its modes. We found that the EMIF is essentially controlled by one 32-bit register. This register provided for many different EMIF configurations. We tried different configurations for the EMIF. After trying several times to send data from the FPGA to the DSP we found a set-up that would transfer all of the accumulated data flawlessly. Once we were sure that the data was being transferred properly all that was left was to simply write the image-processing code for the DSP.

Developing the image-processing algorithm was a very rapid process. It was quick because it was very easy to test and confirm that the code was doing what it was supposed to. For example, finding the vertical lines in the image could be easily verified by outputting the location of the vertical lines the algorithm found and comparing the locations with the actual data. Looking at the data, it is easy to verify that there is indeed a vertical line at the location the algorithm found. The rest of the image-processing algorithm was developed in a similarly by writing some code and then testing. This method proved to be very efficient and allowed us to develop the image-processing code in a few hours.

4.3 System Overview

Currently, the encoder is functional but only works for one scale pattern. Position updates are output through the Code Composer software that came with the DSP. The figure 5.1 shows the current setup.

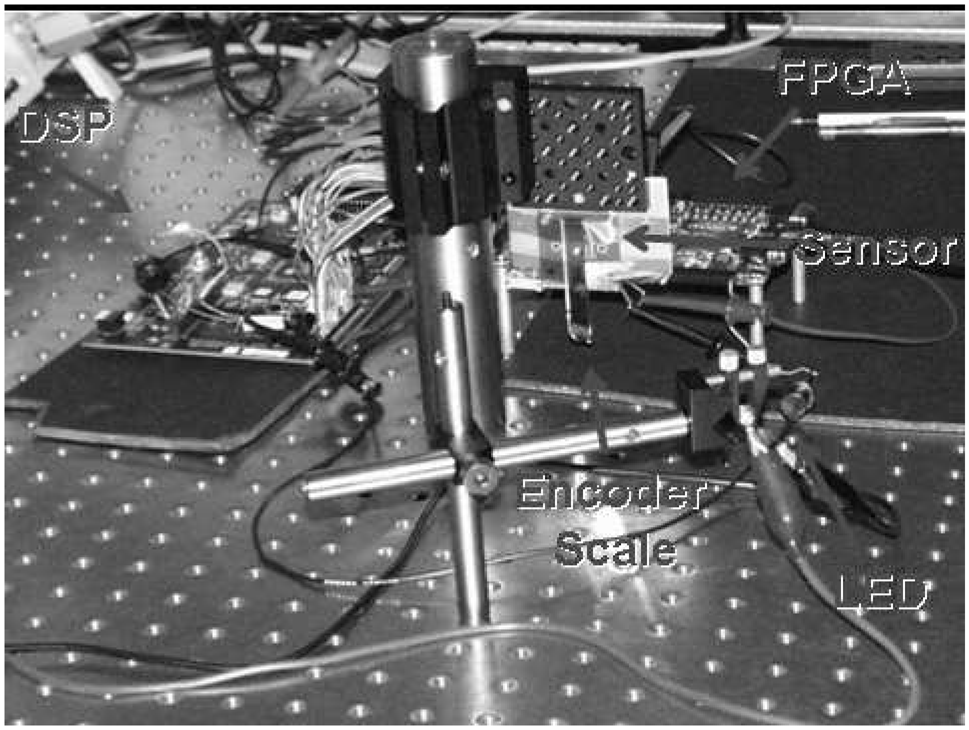


Figure 6: Encoder Setup

The final system that we implemented worked as follows. The DSP would first initialize the registers of the sensor. The FPGA then waits till the start of a new frame before it begins to accumulate data. When a new frame starts the FPGA begins accumulating the 10-bit data coming from the sensor and storing it as 20-bit accumulated data in its internal memory. Once a whole accumulated image is stored the FPGA tells the DSP to perform a block transfer on the data. The DSP then transfers the entire accumulated image into its internal memory. Once the entire accumulated image is stored in the DSP, it then performs the image processing algorithm on the data. Once it completes the image-processing algorithm it outputs the position as a 64-bit floating point number. While the DSP is performing the image-processing algorithm on the first accumulated image, the FPGA is accumulating data for the next image and storing this data. Once the FPGA has the next image accumulated and stored the process starts all over again.

4.4 Summary

This chapter provided details on the research phase, experimentation phase, and an overview of where the project stands now. The next chapters will present project solution and the implementation of the solution. Also, the results, obtained from the methodology described in this chapter, will be presented.

5.0 Results

This chapter will present the results of the methodology explained in the previous chapter. It will also provide technical perspective and precise functionality of the design. We will explain the image processing algorithm, the project solutions, the sensor initialization, and the position updates obtained from the new design of the Leviton Encoder.

5.1 Image Processing Algorithm

To fully understand the reason behind the implementation of our design, it is necessary to have a full understanding of the image-processing algorithm. A detailed explanation of the image-processing algorithm can be found in Appendix A. For our purposes, the actual calculation of the position is the same. However, the technique we use to navigate through the image is different than the one described in Appendix A. Our image-processing algorithm begins navigating through the image by first locating the pixel columns which contain the vertical lines. These columns are determined by summing, along each pixel column, from the top to the bottom of each vertical line. The summation results in a one-dimensional array, from which the vertical lines are located by comparing each element in the array to a threshold value. The threshold value is determined to be half of the maximum value of a summed column. Then the centroid is calculated, as described in Appendix A. After the centroid for each visible vertical line is known, the code bits are located through a known pixel offset for that particular scale. Once the code bits are interpreted, the position is calculated as described in Appendix A. The row markers are unused in this fashion, which allows for a simpler implementation.

5.2 Project Solutions

The LM9637 CMOS active pixel sensor sends out pixel data at a maximum speed of 27 MHz when operating in the Master mode; also known as free-running mode. In order to run the sensor at top speed in the Master mode, the DSP would have to be able to perform the summing operation on the 10 bit pixel data at a rate of 27 MHz. The DSP can perform the summing operation at speeds greater than 27 MHz once that data is in its internal L2 memory. Unfortunately, the EMIF of the DSP is not meant for high-speed real time data transfers. The EMIF of the DSP is capable of transferring data in block transfers at speeds near 100 MHz (Texas Instruments, 2004b). However, using an interrupt to transfer data, one element at a time, is many times slower. This is due to the fact that the DSP requires about 13-15 clock cycles before it responds to an external interrupt (Texas Instruments, 2004b). Sending interrupts at 27 MHz causes the DSP to miss data due to the latency caused by each interrupt. Sending elements one at a time using interrupts results in transfer speeds below 1 MHz. Therefore, it is necessary to perform the summing operation before the pixel data is sent to the DSP or to store blocks of image data and then perform a block transfer of data into the DSP.

Last years MQP used the block transfer as a way of buffering part of the image and then sending the whole block of data at once. They implemented a FIFO, which would take the data coming out of the sensor and store it until half of the image had been transferred. The DSP would then perform a block transfer on that half of the image while the FIFO continued receiving the other half of the image. Although this method would work for this system, in this way, the DSP can only perform the image-processing algorithm on sections of the image because the summing operation on the data cannot be performed until the block transfer has been completed. The internal L2 memory of the DSP is relatively small so that when operating the sensor in full frame mode (640x480). Only about 15% of the image can be stored in memory at any one time before the summing process is performed. This method works well for smaller window sizes (240x200 and lower), where a whole frame of data can be transferred into the internal memory of the DSP. However, if larger window sizes are desired, this process becomes much more complicated.

Initially we were going to take this approach to the DSP data transfer problem. When researching FIFO's and various RAM's that could be used as a data buffer for the DSP, we learned that each column of the data coming from the sensor could be summed into different sized chunks. The part of the column containing the vertical lines would be summed into one value and then each part of the column containing the code bits would be summed up into a separate value for each bit. We researched for a device that could do the summing and then feed the preprocessed image data to the DSP. In the meantime, we also looked at the various FIFO's on the market in case we were unable to locate a device that could perform the desired summing operations.

In order to perform the image-processing algorithm, the whole frame of data should be stored in internal memory of the DSP. Having a full frame of data stored in a memory allows for fast and easy navigation and computation. The DSP, however, cannot hold a full frame, which is 640 x 480 ten-bit pixels or 307200 ten-bit pixel data, in its internal L2 memory, which is 64 kilobytes in size and some of it is designated for program memory. Realistically, this leaves us with about 48 kilobytes for storing the entire image.

Parts of the image will be summed anyway so if the DSP could sum the data as it arrives it would limit number of locations needed to store pixel data. If we summed the parts of the columns that contain the vertical lines, data could be easily accumulated into a 20-bit value, which would be stored as a 32-bit value in the L2 memory of the DSP. For Instance, if the vertical lines were 450 pixels long then only 19840 32-bit words need to be stored after the summing process, as opposed to 307200 16-bit half words without summing. The code bits can also be compressed in a similar fashion without destroying the image data. We know the length of the code bits so we can also sum the code bits along each column. If we have a scale pattern similar to the classical scale pattern shown earlier in figure 2 we will be able to sum each column into three chunks (the vertical lines, top code bits, and bottom code bits); ignoring the row markers. For the classical scale shown in figure 2 if we sum 450 values for the vertical lines part of the image and sum 15 values for each of the two code bit rows, this will result in three 32-bit values per column, as opposed to 480 16-bit values. This summing also reduces the size of our image from 307200 16-bit half words down to 1920 32-bit words, which can easily be stored into the internal memory of the DSP.

5.2.1 Implemented Solution

We discovered that a relatively simple FPGA could perform all the summing functions at the speeds we required. The pixel data of each column coming out of the sensor, at 25MHz—the clock speed provided to the sensor—was accumulated. A threshold value was set to determine the vertical lines. Once the number of pixels that contain the vertical lines was determined by comparing the summed value to the threshold vale, the FPGA was set to output this accumulated data to the DSP. Then the accumulator was reset and the FPGA summed the part of the column that contains the first row of code bits. After the first row of code bits was determined through the same process as the vertical lines, the accumulator was reset for the second row of code bits.

Although the FPGA part was fully functional, the DSP still did not take in the accumulated data at the required speed. We then stored the accumulated values into a RAM of the FPGA. With increased familiarity with the FPGA's, we recognized that the whole image-processing algorithm can be performed on the FPGA. Therefore, we changed our focus to designing logic for the image-processing algorithm. After completing most of the logic for the algorithm, we discovered that 32-bit floating point arithmetic library in VHDL—required for computing the position updates—will require another ten weeks, thus making it impossible for us to use just an FPGA for the whole image-processing algorithm.

On the other hand, the DSP can be easily used to perform 32-bit floating point arithmetic. At this point, we had the FPGA logic store all of the accumulated values in its on chip RAM. As explained earlier, the block data transfer to the DSP is much faster than sending values one at time. Therefore, we performed a block transfer of the FPGA RAM to the DSP over the EMIF. After each transfer the DSP would then perform the image-processing algorithm which is easily written in C code. The C code is in Appendix D and the VHDL code is provided in the CD that accompanies this report.

5.2.2 Ideal Solution

The ideal solution would be to have the FPGA perform the whole image-processing algorithm. This would mean that the encoder will only consist of a sensor and an FPGA, as two main components. Therefore, removing DSP from the encoder, this will significantly reduce the overall cost and complexity of the device. We have proved that the FPGA is capable of performing the whole algorithm, provided that a working 32-bit floating point VHDL library is available. It must be noted that there are such libraries available that can perform floating point operations, however, the lack of time restricted us from getting those libraries to work accordingly with our project.

5.3 Sensor Initialization

It was critical that the LM9637 CMOS active pixel sensor is properly initialized and that we program the sensor registers correctly to obtain desired results. We first used the SNAPS program to initialize the sensor and obtain a VGA output on a computer monitor, which enabled us to visually understand the effects of all the sensors registers. Our next step was to use the 2003 team Visual Basic (VB) program which programs the sensor registers through a Digital I/O card. After successfully programming the sensor with the VB program, we used the DSP to program the sensor. Detailed documentation on sensor initialization can be found in Appendix C.

During our experimentation, we discovered that the sensor must be initialized with a certain register initialization sequence. If the sequence is not followed, the desired results cannot be achieved. This sequence is not mentioned in any of the sensor data sheets. The SNAPS program uses this sequence to initialize the sensor. This sequence is documented in our DSP code and can be seen in Appendix D. After the sequence is followed, other sensor registers can then be programmed in any way.

5.4 System Design

The overall design of the Leviton Encoder is significantly changed from last year. For the first time, the FPGA is used to perform most of the image-processing algorithm and the DSP is being used to perform simple arithmetic. This section will give a detailed description of the FPGA introduced in the device and the new system block diagram.

5.4.1 FPGA

The FPGA was carefully selected for its functionality, low cost, and our familiarity with Xilinx products. Following are the main features of the FPGA in the current system.

- Spartan XC3S200
  - 200k system gates
  - 326MHz system clock
  - 216K bits RAM
  - 173 I/O pins
  - Evaluation board $100

        - on board Flash memory
    - chip = $ 16

The Spartan XC3S200 has plenty of system logic gates and internal RAM for the task at hand. There are sufficient amount of I/O pins needed and the evaluation board has a Flash Memory, which can be programmed through a JTAG connection. Therefore, the board does not have to be programmed at every startup. Moreover, the cost of the board and the chip is more than acceptable, when compared to the DSP board, which alone costs over $400.

5.4.2 Encoder Design

The Leviton Absolute Encoder now relies on FPGA to perform most of the image-processing algorithm. The following is a high level block diagram of the new Leviton Encoder:

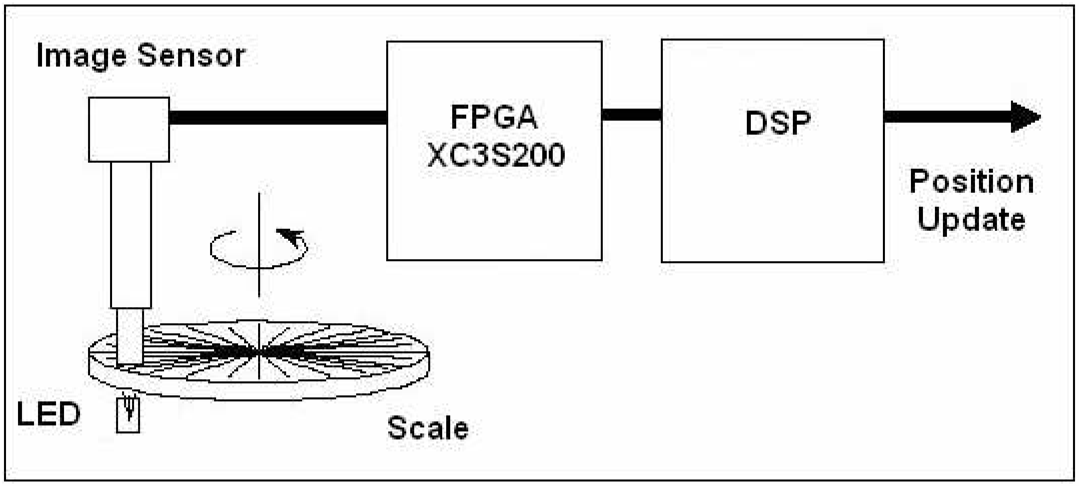


Figure 7: New Block Diagram

Ideally, the encoder will only have the Image sensor and the FPGA. However, in the above diagram, the DSP is utilized to obtain position updates. Also, DSP provides various options on collecting position updates and presenting it in a data file, which can later be used in a spread sheet program for graphical analysis.

5.5 Position Updates

We obtained three different position updates under same conditions. One was the encoder just placed on the table and the second, in which the encoder scale was moved eighteen times with about 50 microns increment every time. The condition in which these first two results were obtained was the worst case condition, in terms of window size. Following are the condition:

- Sensor Master Clock = 25MHz
- Window = 640 x 480
- Row Delay = 8h
- No lens

The frame rate under these conditions is 63.2 frames/second. The figure 5.3 shows 500 position updates obtained with the encoder just placed on a table.

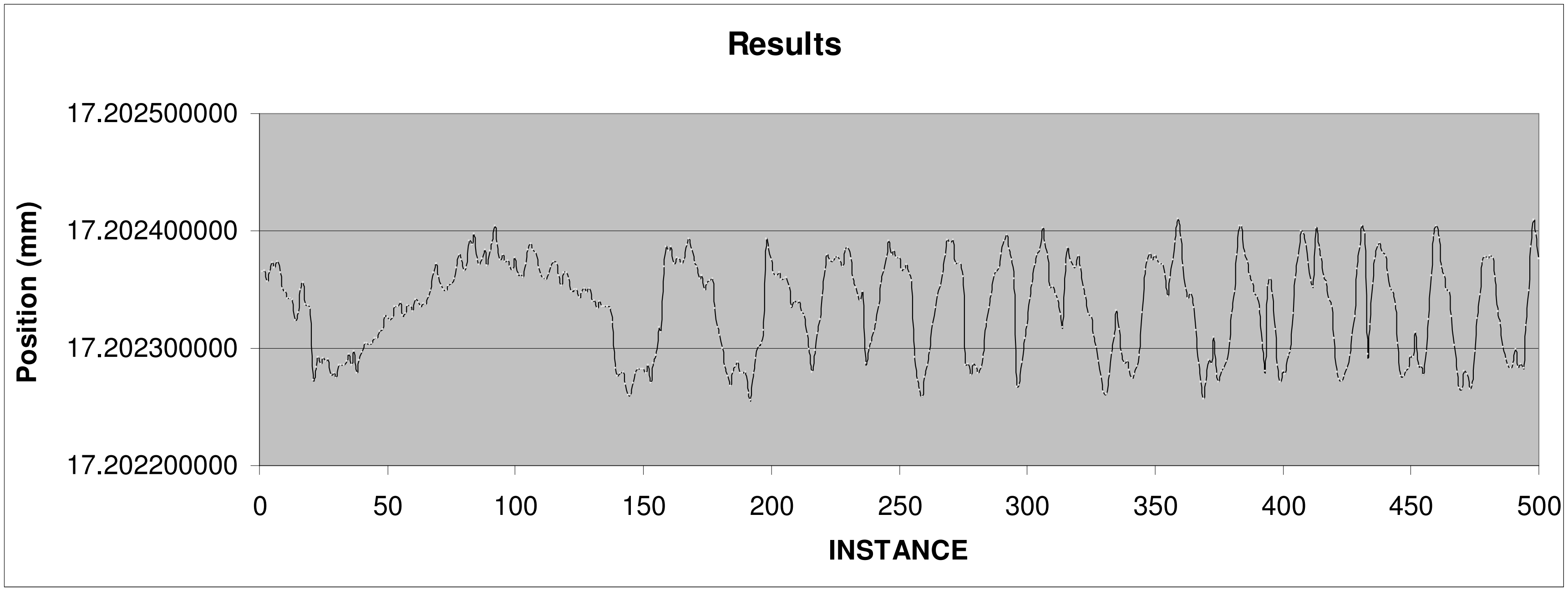


Figure 8: Noise Position Updates

The previous chart shows the 500 position values on a 100nm division. Since the setup was not isolated from vibration and noise of other equipment on the table, the resulting position is oscillating. The figure 5.4 shows the position updates as the encoder was moved vertically on the sensor 18 times by about 50 microns.

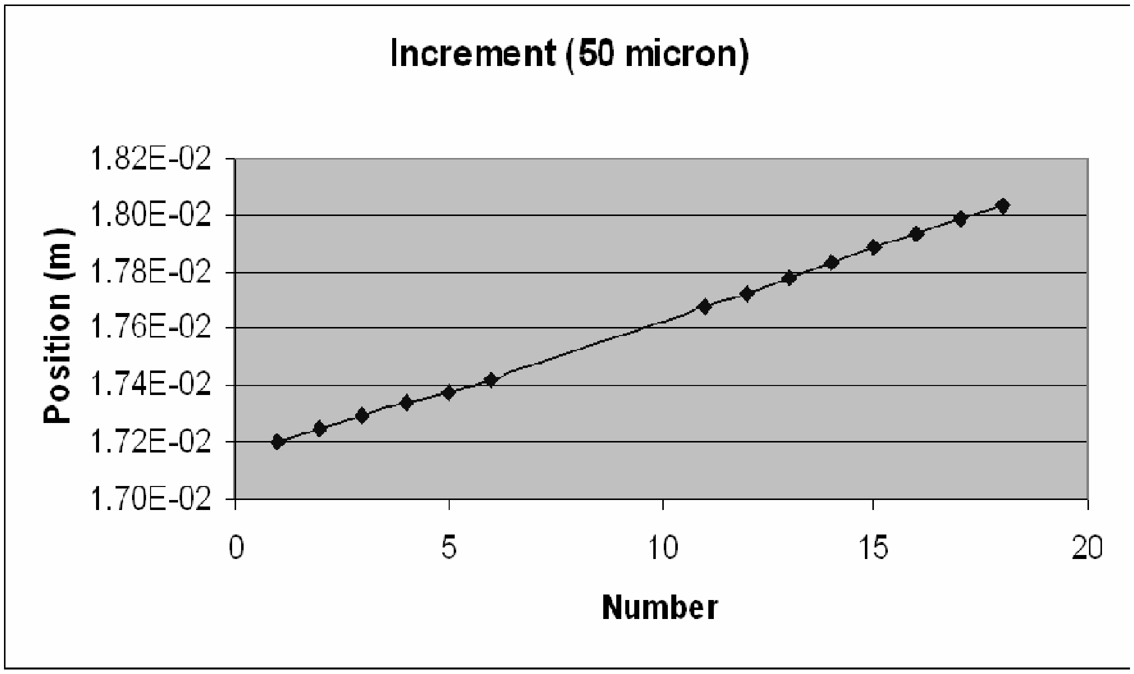


Figure 9: Increment Position Updates

It must be noted that each increment was approximately 50 microns. The four data values, number seven through eleven, not shown in the above graph were excluded because of their inaccuracy, solely due to the setup of the test. The next eight values on the graph prove that, in fact, the setup of the encoder scale on the sensor caused inaccurate results in those four increments. To further prove that the setup was to be blamed for skewed values, the VGA output was obtained which revealed that the code bits were cutoff, thus, the sensor was sending incomplete pixel data, in terms of the encoder scale.

Our third test was with a small window size of 128 x 480. Due to lack of time, we were only able to run this test once. With this window size, the frame rate, calculated by the Excel Spreadsheet provided in the CD, is 315 Hz. We were able to get 315 position updates in one second. This test further proved that the encoder settings can be changes to obtain even faster frame rate.

5.6 Summary

The development of our version of the Leviton Absolute Optical Encoder involved tremendous amount of research. We relied on 2003 MQP report to obtain for sensor initialization, which was extremely well documented. We found that an FPGA can be used to perform the whole image-processing algorithm, further making the encoder more marketable. The final results of our project proved that the encoder is capable of reliably providing position updates.

6.0 Conclusion and Recommendations

In this chapter, we conclude our results and discuss the current status of the Leviton Encoder. For future development of the Leviton Encoder, recommendations and their explanations are also provided. These recommendations will enhance the current design and make it more marketable.

6.1 Conclusion

Our implemented solution solves the project problem. For the first time, the Leviton Encoder is able to provide position updates using the LM9637 CMOS active pixel sensor. This new encoder design also proved that just an FPGA can be used to perform the image-processing algorithm, as mentioned previously in our ideal solution. Also, this new design reduces the overall cost and complexity of the design and increases the marketability of the Leviton Encoder.

6.2 Recommendations

The following are the recommendations for the next MQP group to further improve the encoder design and also to make it more marketable.

- Use FPGA to fully implement the image-processing algorithm

Currently the encoder relies on the DSP to perform the image-processing algorithm. We found that the position updates could be obtained, at a much faster rate, if the image-processing algorithm was performed entirely in an FPGA. We could not implement the image-processing algorithm in this manner due to the lack of floating point library in VHDL, as discussed previously in this report. Next years MQP group should work on developing a way to perform floating point math on the FPGA.

- Incorporate all scale patterns and window sizes in the current system

Once the image-processing algorithm is successfully implemented for one scale pattern and window size, work will be need to done to accommodate the other scale patterns and window sizes. This will involve creating several modes of operation including a high accuracy mode. In this mode, the sensor would use the full window size and then three to four position updates will be averaged together to output one position update. Also, there may be a high speed mode in which a very small window size is used and extreme accuracy is not needed. For each scale, the image-processing algorithm significantly changes, therefore, it the logic should be designed accordingly.

- Develop a way to initialize the sensor without using the DSP

With the FPGA performing the image-processing algorithm, there needs to be a way to program the sensor without using the DSP, thus completely eliminating the DSP from the system. Last years MQP team (2003) began development of FPGA logic that would initialize the sensor. They stopped this development when they found out that the DSP could easily initialize the sensor. Last years work can be used as a starting point to develop FPGA logic, which programs the sensors registers. Other solutions involve using a PIC microcontroller to communicate with the sensor.

- Achieve a Rate of 2000 position updates per second

The current encoder set-up was only tested at 60 position updates per second and 300 position updates per second. Unfortunately due to lack of time we were unable to test at higher speeds. The current set-up should be tested at the minimum allowable window size to see the top speed that can be reached with this encoder design. With the FPGA performing the image-processing algorithm and with the correct sensor settings, achieving a speed of 2000 position updates per second should be trivial.

- Create a video output for the encoder for alignment purposes

Currently, there is no easy way to check the alignment of the sensor relative to the scale. The alignment is very important for the image processing algorithm to work properly. There should be an easy way to verify that the sensor is indeed aligned properly over the scale. One way of doing this would be to turn the sensors output into a video signal and then display this onto a computer monitor or television screen. This would allow for quick and easy alignment check.

- Fully implement the design on a PCB

Once the system has been fully developed using the various development boards, the design should be moved onto a PCB for further prototyping. The entire PCB design should be as small as possible and ideally fit into a 2-inch cube. If the DSP is removed from the system the entire PCB should fit easily into a 2-inch cube. The only way the encoder would not be able to meet this requirement is if the connectors which communicate the position updates to the host are big. A universal serial bus (USB) connector will be a good choice. It is a small connector and is compatible with almost all the computers, nowadays. Unfortunately, this may not be the standard for encoders in the market today. The connectors and PCB design will need to be researched in detail to meet current standards.

References

Fowler, R., Koski, A., Nguyen, H., (2001). Optical Encoder Technology Development. Greenbelt, MD.

Garber, Stephen J., Launius, Roger D., (2002). A Brief History of the National Aeronautics and Space Administration. Retrieved August 17, 2004, from the World Wide Web: http://www.hq.nasa.gov/office/pao/History/factsheet.htm

Goddard Space Flight Center (GSFC)., (1998). Goddard Space Flight Center. Retrieved August 17, 2004, from the World Wide Web: http://www.gsfc.nasa.gov/gsfc/about_gsfc.html

Harvey, Eric., Cate, Austin., (2003). Leviton Absolute Optical Encoder Development. Greenbelt, MD.

Jaeger, B., Tremelling, G., Wartman, R., (2002). Improving the Leviton Absolute Optical Encoder. Greenbelt, MD.

Leviton, Douglas B., (2000). Absolute Position Encoder Using Pattern Recognition. Retrieved August 17, 2004, from the World Wide Web: http://www.nasatech.com/Briefs/June00/Gsc13703.html

Leviton, Douglas B., (2003). Image processing for new optical pattern recognition encoders. NASA Goddard Space Flight Center.

National Semiconductor (2000). LM96xxEVAL-KIT User Guide & Reference Manual. Retrieved August 19, 2004, from the World Wide Web: http://www.national.com/appinfo/imaging/files/EvalGuidev3.pdf#page=1

National Semiconductor, (2002a). LM9637 Camera Headboard. Retrieved August 19, 2004, from the World Wide Web: http://www.national.com/appinfo/imaging/files/LM9637_HEADBOARD.pdf#page=1

National Semiconductor, (2002b). LM9637 Monochrome CMOS Image Sensor VGA 68 FPS. Retrieved August 19, 2004, from the World Wide Web: http://www.national.com/ds/LM/LM9637.pdf#page=1

NTB, (2000). Who's Who at NASA Douglas B. Leviton, Optical Physicist. Retrieved August 17, 2004, from the World Wide Web: http://www.nasatech.com/NEWS/June00/ntb.june00_leviton.html

Texas Instruments, (2004a). Code Compose Studio Getting Started Guide v3.0
. Retrieved August 19, 2004, from the World Wide Web: http://www-k.ext.ti.com/SRVS/CGI-BIN/WEBCGI.EXE/,/?St=171,E=0000000000007042181,K=6978,Sxi=1,Case=obj(1720 7

Texas Instruments, (2004b). TMS320C6000 Peripherals Reference Guide. Retrieved August 19, 2004, from the World Wide Web: http://www-k.ext.ti.com/SRVS/CGI-BIN/WEBCGI.EXE/,/?St=93,E=0000000000007041949,K=6851,Sxi=1,Case=obj(2343

Texas Instruments, (2004c). TMS320C6711, TMS320C6711B, TMS320C6711C Floating-Point Digital Signal Processors. Retrieved August 19, 2004, from the World Wide Web: http://focus.ti.com/lit/ds/symlink/tms320c6711.pdf

APPENDIX A: Image Processing Algorithm

Image processing for new optical pattern recognition encoders

Douglas B. Leviton
Goddard Space Flight Center
Code 551
Greenbelt, MD, 20771

1. ABSTRACT

An all new type of absolute, optical encoder with ultra-high sensitivity has been developed at NASA's Goddard Space Flight Center.[i] These position measuring encoders are unconventional in that they rely on computational pattern recognition of high speed, electronic images made of a moving, backlit scale which carries absolute position information of either linear or rotary format. The pattern recognition algorithms combine edge detection, threshold level sensing, spatial compression, and centroiding along with fault recovery through scale image defect detection. Details of the encoder scale patterns and their design rules and the image processing algorithm which gives these encoders their unique and unparalleled performance characteristics are discussed.

Keywords: NASA absolute optical encoder; pattern recognition; image processing; algorithm

2. BACKGROUND

NASA's new pattern recognition encoders use a combination of mature technologies including high accuracy microlithography, optical projection, charge-coupled-device (CCD) array image detection, and simple image processing. In most cases, a fixed, light emitting diode (LED) backlights a microlithographically patterned scale attached to the moving part of a positioner, and a fixed CCD array camera records images of the portion of the scale within the CCD's field of view. The CCD's electronics rapidly digitize the images of the scale pattern into a memory buffer. A pattern recognition algorithm operates on the image buffer contents to derive a numeric position based on the known pitch of the glass scale pattern. This paper focuses on scale design flexibility and on the details of that algorithm.

Small format, monochrome CCD's are ideal for use in these encoders since images can be read out at the highest possible frame rates and the amount of image data required to get accurate encoder readings can be minimized. The LED's brightness is automatically controlled as part of the image processing algorithm in order to optimize image density to yield the most valid, accurate encoder readings.

3. SCALE PATTERN BASICS AND DESIGN RULES

A section of a linear encoder scale pattern is shown in Figure 1. A typical image of the moving scale pattern as seen by the fixed CCD camera is shown in Figure 2. The narrow vertical bars in the top three quarters or so of the pattern are called fiducials. The long dimension of those scale features is arranged perpendicular to the direction of motion in a strictly periodic fashion. The large horizontal rectangles at the bottom are called row markers. These are used by the pattern recognition software to a) determine the vertical locations of all other image features and b) to define the width and location of the window over which fiducial centroids will be computed. The small squares just above the row markers are called code bits. These are arranged in a format which represents a sequential binary code to uniquely identify each fiducial, thus making the encoder absolute versus incremental. A row marker together with its associated code bits and fiducial are called a code group

A vertical line drawn down the middle of the image (in line with the arrow at the bottom of Figure 2) would demark the center column of pixels on the fixed CCD array. It is the centroid position of the image of each fiducial with respect to that pixel column which allows position to be determined with ultra-high sensitivity. The horizontal motion of each fiducial's centroid as seen by the CCD is linear with actual motion in that direction.

The NASA encoder offers extreme design flexibility because the scale pitch (the physical distance between adjacent fiducials for a linear encoder or the angular distance for a rotary encoder) and optical magnification can be chosen to suit the requirements of essentially any application. They are also chosen such that there will always be at least two and sometimes

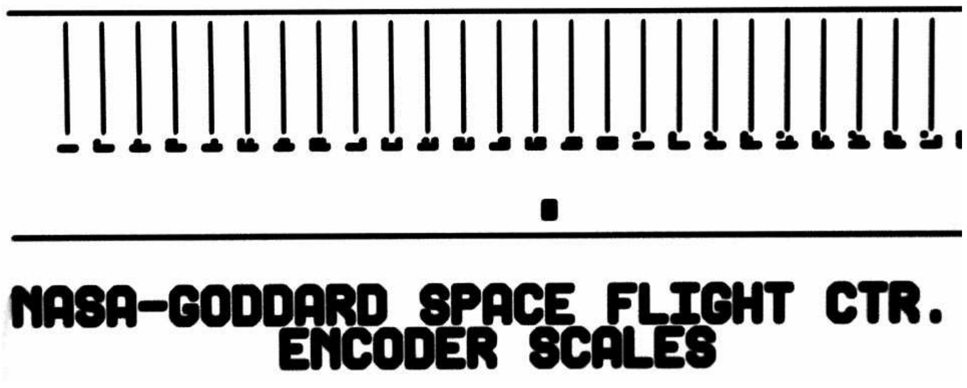


Figure 1 – Negative of photo of typical section of NASA linear, absolute, optical encoder scale (Type I – 2 mm pitch); sequential code groups can be seen but not crisply due to hand-held closeup

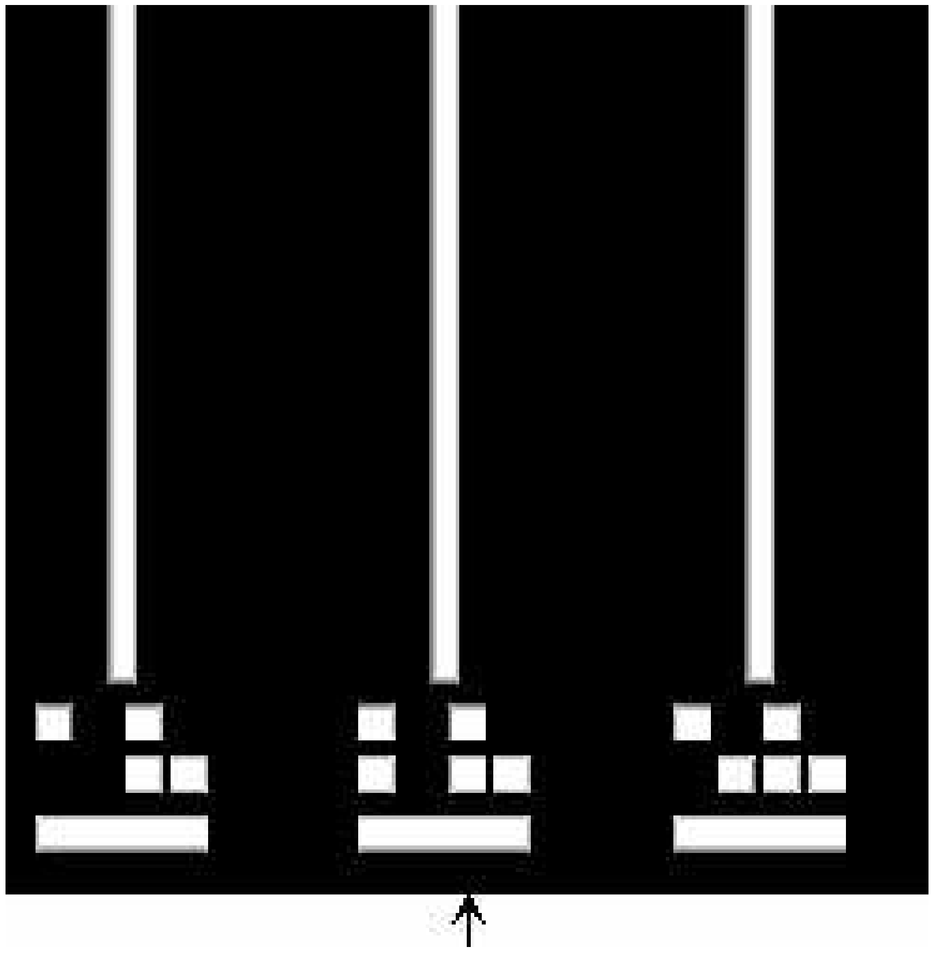


Figure 2 -- Sample image of absolute encoder scale pattern as seen by fixed CCD camera; arrow at bottom points to center column of pixels

just barely three entire code groups in each image. The encoder software is aware of the pitch and continuously adjusts its perceived image scale (microns per pixel or degrees per pixel) based on measured differences in locations of fiducial image centroids measured in pixels on the CCD. This operation requires the presence of at least two code groups in each image. Generally, such adjustments are entirely miniscule but this practice tends to average out any errors in placement of individual fiducials on the scale.

For an encoder using a CCD array having roughly 200 pixel columns, a few rules of thumb guide the selection of scale pitch and imaging optics. Modified rules apply to different CCD formats. The first rule of thumb is that the encoder's resolution will be approximately 1/10,000th of the scale pitch. A smaller pitch provides extra resolution for free but requires higher magnification from the imaging optics in front of the CCD. The second rule of thumb is that the magnification should be nearly equal to the physical width of the CCD array divided by thrice the scale pitch. For optimal centroiding of fiducial images, the physical width of the fiducial on the scale should be such that its image covers 4 to 6 pixel columns on the CCD. A few examples illustrate these rules of thumb for linear and rotary encoders.

3.1 Linear Scale Design Examples

An encoder with a resolution of 5 nanometers (0.005 microns) is required. Thus, the scale pitch should be no larger than 50 microns and the optics must image the scale onto a 2.6 mm wide CCD with a magnification = 2.6 mm / (3 x 0.05 mm) = 17X. A 20X microscope objective with and scale pitch tweaked to 43 microns would work. Three rows of code bits containing four bits each would encode a range of 4096 x 0.043 mm = 176 mm (7 inches). This resolution is better than that of several commercially available distance measuring interferometers.

An encoder with a resolution of 0.1microns (100 nm or 4 microinches) is required. The scale pitch needs to be no larger than 1 mm, and the optics must image the scale onto a 2.6 mm wide CCD with a magnification = 2.6 mm / (3 x 1 mm) = 0.87X. This is a particularly interesting example in that the needed magnification is very close to unity. This encoder would be designed for "shadow" mode, where a scale with 900 micron pitch is backlit with "collimated" light and simply casts a shadow on the CCD directly behind it with a small gap. This configuration is compact, simple, has lowest parts count (and no optics per se), and works naturally at unit magnification. Three rows of code bits containing four bits each would encode a range of 4096 x 0.9 mm = 3,686 mm (145 inches or 12 feet). To implement this range, multiple scales are joined together at their ends, with stitching errors calibrated out. Scales of this pitch are exceptionally immune to scale damage or contamination. Three rows of five code bits would extend the range to 32,768 x 0.9 mm = 30 m (100 feet)!

An encoder with a resolution of 0.0025 mm (2.5 microns or 0.0001 inches) is required. The encoder in the previous example is such a cheap, simple, and convenient configuration that the application in this example should probably be built that way and extra resolution provided just ignored. But we go through this example anyway as there might be practical reasons why a shadow mode encoder could not be used. For a 2.6 mm wide CCD, a scale pitch of 25 mm (1 inch) and optics with a magnification 0.033 would be used. Here, the scale pitch has grown to a degree that the camera views an area on the scale 75 mm (3 inches) on a side, which might pose packaging limitations for some applications. Yet, this design provides enormous travel, exceptional resolution, and is extremely immune to scale damage. It lends itself to an etched metal band scale fabrication technique or other. Intermediate scale pitches such as 5 to 20 mm also lend themselves to etched metal band scale fabrication giving resolutions from 0.5 to 2 microns with scale heights from 15 mm to 60 mm, respectively. For completeness, we can readily see that three rows of code bits containing only four bits each would encode a range of 4096 x 25 mm = 102 m or the length of a football field!

3.2 Rotary Scale Design Examples

A full revolution encoder with resolution of 0.01 arcseconds (3 x 10^{-6} degrees or 50 nanoradians) and scale diameter less than 300 mm (12 inches) is required. Having a power of two for the number of code groups around the perimeter of the code track is convenient but not essential. 4096 code groups gives 0.0879 degrees per code with a resulting resolution of 9 x 10^{-6} degrees which is not good quite enough. 15 code bits per group in three rows of five would certainly do the trick with 32,768 possible codes. For convenience, one might wish to have the code groups at easy to remember intervals like 0.02 degrees or 18,000 groups around the track, giving a resolution just better than the requirement at 2 x 10^{-6} degrees. We have enough information to compute physical separation of code groups on the scale and to determine the required optical magnification. A 300 mm diameter track has a perimeter of 300 mm x π = 942 mm. 942 mm divided by 18,000 codes = 0.052 mm per code. The optics will be designed to image the scale onto a 2.6 mm wide CCD with a magnification of 16X.

A full revolution encoder with a resolution of 0.2 arcseconds (5.6 x 10^{-5} degrees) is required and the scale diameter is required to be less than 75 mm (3 inches). 2048 code groups gives 0.176 degrees per code with sufficient resolution of 1.8 x 10^{-5} degrees (<0.1 arcseconds) — the equivalent of a 24 bit conventional absolute encoder . The physical separation of code groups on the scale will be 75 mm x π / 2048 = 0.115 mm. The required optical magnification will be 2.6 mm / (3 x 0.115 mm) = 8X. This design could just as easily be scaled down with the same resolution to use a 50 mm (2 inch) diameter scale with 2048 codes, a separation of about 0.075 mm between codes, and a magnification of roughly 12X.

Suppose a sector encoder which covers a range of only +/- 10 degrees is required and the encoder scale can be mounted with respect to the rotation axis at a radius as large as 500 mm (20 inches). What different encoder resolutions can be gotten and what would their packages look like? Let us look at a few different scale pitches and allow those to prescribe magnifying optics and achievable resolution.

We first look at a scale with code groups 0.05 mm apart. From previous examples, the optics must provide a magnification around 16X At a radius of 500 mm, the angular separation of codes will be 0.0057 degrees and the resolution will be about 6 x 10^{-7} degrees (0.002 arcseconds). A microscope arrangement or custom-designed, molded lens system could be used, depending on package constraints. Now, let us consider a scale pitch of 0.9 mm for a shadow mode design. At a radius of 500 mm, the angular separation of codes will be 0.103 degrees and the resolution will be a very respectable 1 x 10^{-5} degrees (0.04 arcseconds). The arc length for a scale covering 20 degrees at a radius of 500 mm is roughly 175 mm. Several scales at either pitch could be economically recorded on a single, 200 mm square master plate. Other intermediate pitches could be used as well, for example, a pitch of 0.180 mm would suggest a magnification of 5X and a resolution of 2 x 10^{-6} degrees (0.007 arcseconds).

4. PATTERN RECOGNITION ALGORITHM

Getting a single encoder reading is a sequence of the following steps: 1) exposing the CCD while concurrently flashing the LED and reading out the resulting image into a memory buffer with 8 bits of brightness resolution (one byte per pixel); 2) finding row markers in the image; 3) deciphering the pattern of code bits associated with each row marker; 4) computing the centroid of the fiducial associated with each row marker; 5) converting the centroid and fiducial identity information to a position; and 6) adjusting the exposure conditions (exposure time and LED brightness) for the next CCD exposure.

The first image processing routine finds the bottom and top edges, and left and right ends of all row markers it can find in each encoder scale image using a simple edge detection process. These edges allow us to navigate through the image in the remaining processing steps. Row marker's whose ends are conclusively identified are marked as valid for those steps. To find these edges, entire row contents are summed, starting at the bottom of the image, and compared with the sum from the previous row (clearly, row markers can not occupy the first row of image pixels). When the new row sum exceeds the preceding sum by some threshold value, the current row is taken as the row which defines the bottoms of the markers. The process is then continued until a row sum dips below the preceding sum by that same threshold value, where the preceding row defines the tops of the markers (see Figure 3).

The middle row of the markers is taken to be the average of the row numbers for the top and bottom of row markers. This row is now searched for the ends of row markers. Starting at the beginning of the row, each pixel value is compared to a single pixel brightness threshold value of about 80 (where maximum brightness for any pixel is 255) looking for low brightness to high brightness transitions to denote the start column of row markers and vice versa to denote the stop columns. The length of each row marker must be within some tolerance of the expected value for the row marker to be considered valid. A more robust treatment would use all rows between top and bottom of row markers, which would help reduce the effects of scale damage or contamination in the row marker area, but experience has generally shown this to be unnecessary.

Relative pixel grid offsets, well known by design, from the now-known middle and ends of all valid row markers are used to pinpoint image pixels where each code group's identifying code bits are centered. By convention, code bits are arranged in rows with the least significant bit (LSB) at the bottom left and the most significant bit (MSB) at top right. Referring again to Figure 2, there are two rows of four bits defining 256 (2^8) possible fiducials, although any other arrangement will work. (The range of a linear encoder is simply the number of fiducials times the scale pitch.) For each code group, the group's identity is built up from 0 as a running sum of binary weighted contributions of each code bit in the grid. If the brightness of the pixel at a code bit location exceeds the single pixel brightness threshold, then that bit is set and contributes to the running sum by its binary weighted value. Normally, only individual pixels are considered in order to minimize software execution time. In Figure 2, in which there are only two rows of code bits, the leftmost code is deciphered as 96 (bits 2, 3, 4 and 6 are set so the

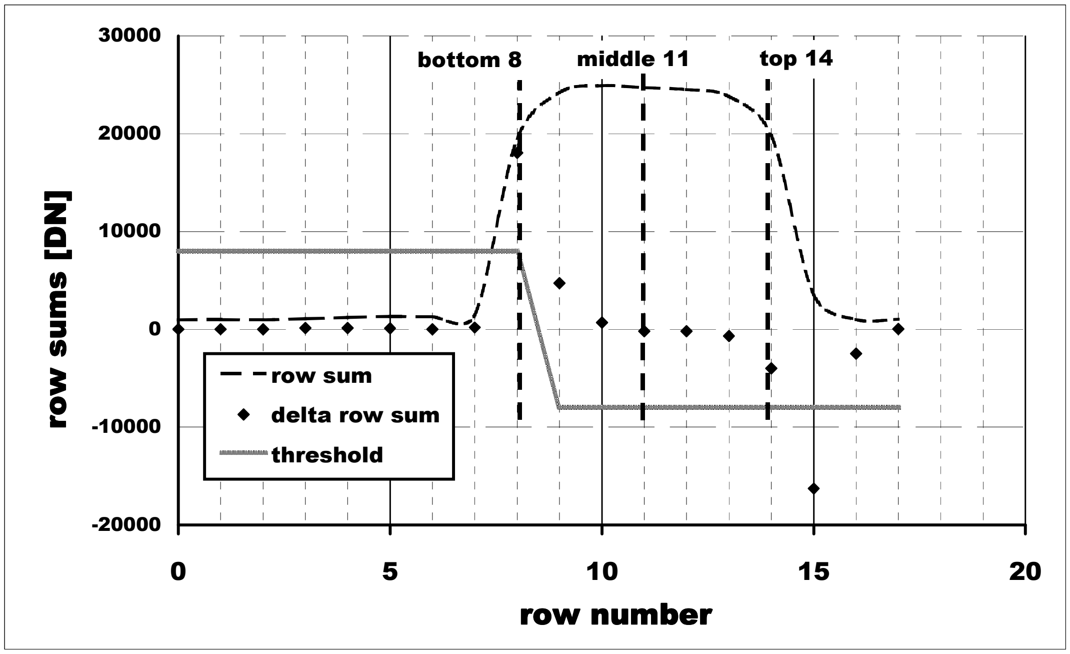


Figure 3 – Changes in row sums which are greater in magnitude than a selected row sum threshold indicate top and bottom edges of row markers

code identity is 4 + 8 + 16 + 64), the center code is 97 (bits 0, 2, 3, 4 and 6 are set so the code identity is 1 + 4 + 8 + 16 + 64), and the rightmost code is 98 (bits 1, 2,3 , 4 and 6 are set so the code identity it 2 + 4 + 8 + 16 + 64). These code identities (***n***, ***n*** + 1, ***n*** + 2) are used later to scale the encoded position result.

An important function of the algorithm is to check to make sure that any code groups identified in each image are sequential. Otherwise, an incorrect position might ultimately be reported. This could happen, for instance, if a particle were occluding the central part of a code bit on the scale which should appear transparent (bright) but instead appears dark and gets sensed below threshold. The deciphered code value would be wrong. Meanwhile, if other code values in the image on either side of this one are okay, this one can either be ignored or re-examined to see if it would be sequential but for one bit and thus could be validly recoded. Alternately, at the expense of execution speed, the entire code bit group could be reprocessed treating the average brightness of the entire cluster of pixels surrounding each single code bit pixel to represent bit brightness compared to the single pixel threshold. Such a code group could also be marked as suspect in an operational lookup table. It is clear that this encoder, through its image processing algorithm, has numerous possible levels of fault detection and recovery.

As motion of the positioner occurs, images of scale fiducials traverse the fixed field of view of the CCD horizontally in a fashion which is linear with that motion. It is the left-to-right centroid of each fiducial that encodes position with a resolution given by the noise in determining those centroids to small fractions of a pixel column. To first order, the image brightness for the fiducials does not vary from row to row but does vary in a characteristic fashion from column to column, similarly for each fiducial. Thus, since the fiducials are parallel to the pixel columns, the first step in computing the light centroids is to sum together the entire brightness contents of each column for all rows starting just above the code bits into a one dimensional array with a number of elements equal to the number of CCD pixel columns. This is what we call a "y-squunch" and it builds peak signal of the centroid operand from that attainable from a single row to a peak signal well over 100 times higher, vastly improving the statistics and execution speed of the centroid computation, and giving the encoder its ultra-high sensitivity. The resulting peak column sum is of order 30,000 DN (digital numbers).

The centroid $\boldsymbol{\sigma}$ of each fiducial image is computed within a window centered on its respective row marker. The width of the window is a parameter which can be specified depending on the column extents of the brightness distribution of the fiducials. Too wide a window reduces resolution while too narrow a window introduces random amplitude, small scale non-linearity errors. Each brightness value operand in the centroid computation is radiometrically corrected by subtracting an average background brightness value taken from an area between fiducial computation windows. The background value represents the sum of the CCD's electronic readout bias, dark signal, and signal due to any detector illumination not related to the encoder scale image itself. Typically, uncertainty in computing fiducial centroids is about 0.005 pixels. Using derived centroids for at least two fiducials and the known scale pitch $\boldsymbol{\delta}$, the algorithm can easily derive image scale $\boldsymbol{\lambda}$, in "microns per pixel" or "degrees per pixel," for example.

Finally, defining absolute "zero position" to occur when the 0th code group is exactly centered on the center pixel column of the CCD, absolute position of the scale with respect to the fixed CCD camera can now be computed by knowing which fiducials are in the scene, knowing how much their centroids are displaced in pixels from the center column of pixels, and knowing the image scale. Since only one position is being encoded, the readings from each fiducial taken independently must necessarily agree. A fiducial's centroid $\boldsymbol{\sigma_n}$ is considered to be negative if it lies to the left of the center column of pixels and positive if it lies to the right. For each fiducial $\boldsymbol{n}$, position $\boldsymbol{r_n}$ is given by the expression:

$$\boldsymbol{r_n = \delta \cdot n - \sigma_n \cdot \lambda_n} .$$

[Eq. 1]

The first term is a coarse offset accounting for the fact that the CCD may see other than the 0th code. The second term is a fine position offset for that fiducial from the CCD coordinate origin defined by the center pixel column. Equation 1 can be rearranged to show that position is, in fact, directly proportional to scale pitch $\boldsymbol{\delta}$, since $\boldsymbol{\lambda}$ is derived from $\boldsymbol{\delta}$ in the first place.

$$\boldsymbol{r_n = \delta[n - \sigma_n / (\sigma_{n+1} - \sigma_n)]}.$$

[Eq. 2]

In this form, position is equal to scale pitch times a fiducial cycle offset from zero equal to some integer number of cycles plus some phase within a cycle. The reading returned by the encoder algorithm is the average of positions determined for all fiducials in the scene.

Image brightness for each CCD exposure is optimized to provide the highest signal-to-noise ratio for centroid computations by adjusting CCD exposure time and LED brightness to maintain a peak column sum just below digital saturation. Source brightness is adjusted through software control of an 8 bit digital-to-analog converter to moderate LED drive current. CCD exposure time in integer milliseconds is sent as an adjustable parameter to the image collection routine. If the peak column sum goes outside of a certain range, LED brightness is first adjusted, also within some range of brightness values. If the requested brightness value is out of range, exposure time is increased or decreased as needed and the brightness value is set to mid-range.

5. ALGORITHM APPLIED TO EXAMPLE IMAGE

Let us step through the entire image processing algorithm for our sample image whose format is 165 rows by 192 columns. In Figure 4a), the algorithm has found the bottoms of row markers at row 8, the tops at row 14, and the centers at row 11. In Figure 4b) it finds the start of row marker 1 at column 5 and the end of row marker 1 at column 40. Similarly, it finds the starts and ends of row markers 2 and 3 at columns 74, 108, 143, and 178, respectively. The row markers are calculated to have lengths of 36, 35, and 36 pixels, respectively. The algorithm verifies that the apparent length of each row marker is between 32 and 40 pixels, and considers them all to be valid.

In Figure 4c), through known row offsets (9 between row marker and first row of code bits and 9 between rows of code bits) and column offsets (3 from start of marker to first code bit column then 9 between subsequent code bit columns) specific to this scale pattern and nominal optical magnification of the encoder optics, the pixels for code bits for each row marker are located at (column, row) coordinates given as follows:

Row marker 1: {0-7: (8, 16) (17, 16) (26, 16) (35, 16) (8, 25) (17, 25) (26, 25) (35, 25)}
Row marker 2: {0-7: (77, 16) (86, 16) (95, 16) (70, 16) (77, 25) (86, 25) (95, 25) (104, 25)}
Row marker 3: {0-7: (146, 16) (155, 16) (164, 16) (173, 16) (146, 25) (155, 25) (164, 25) (173, 25)}.

As we saw earlier, the algorithm determines that the three fiducials in the scene are numbers 96, 97, and 98 by examining the brightnesses of the bits with respect to a single pixel threshold value to see if each bit is "set" or not. It realizes that these values are, in fact, sequential and so considers all three row markers to still be valid.

Next, in Figure 4d), the algorithm identifies that the fiducials occupy rows 34 to 165 (132 inclusive). It now creates a one dimensional array of "super-columns" by adding the row contents for all of the latter rows together. In Figure 5, the regular curve is a plot of the brightness distribution of the raw super-column array. An omnipresent background value (the average value in a region between fiducials -- exaggerated in Figure 5) is subtracted from the raw distribution to yield a distribution which is due to scale image light only. The adjusted brightness distribution (bold curve in Figure 5) is the data source for computing fiducial centroids. The peak column value is stored for use in adjusting exposure for the next image to be taken.

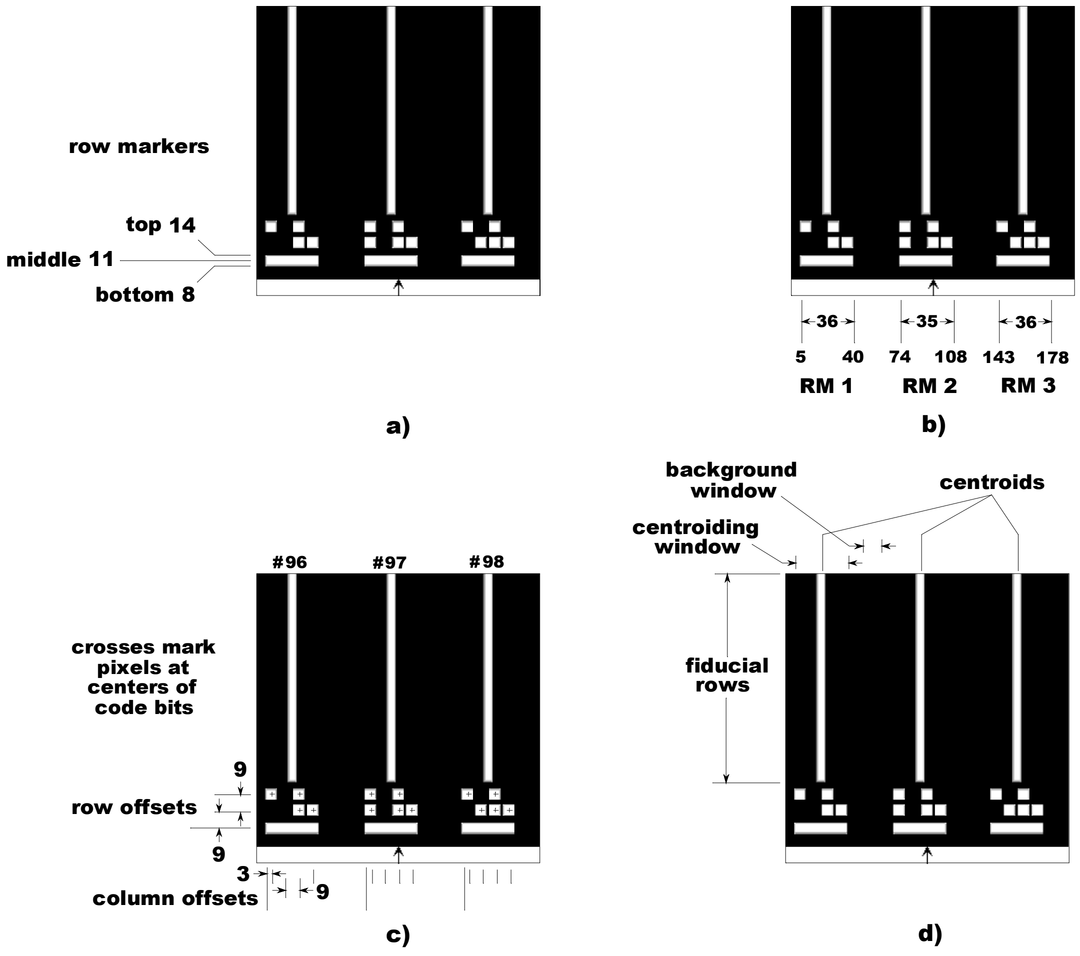
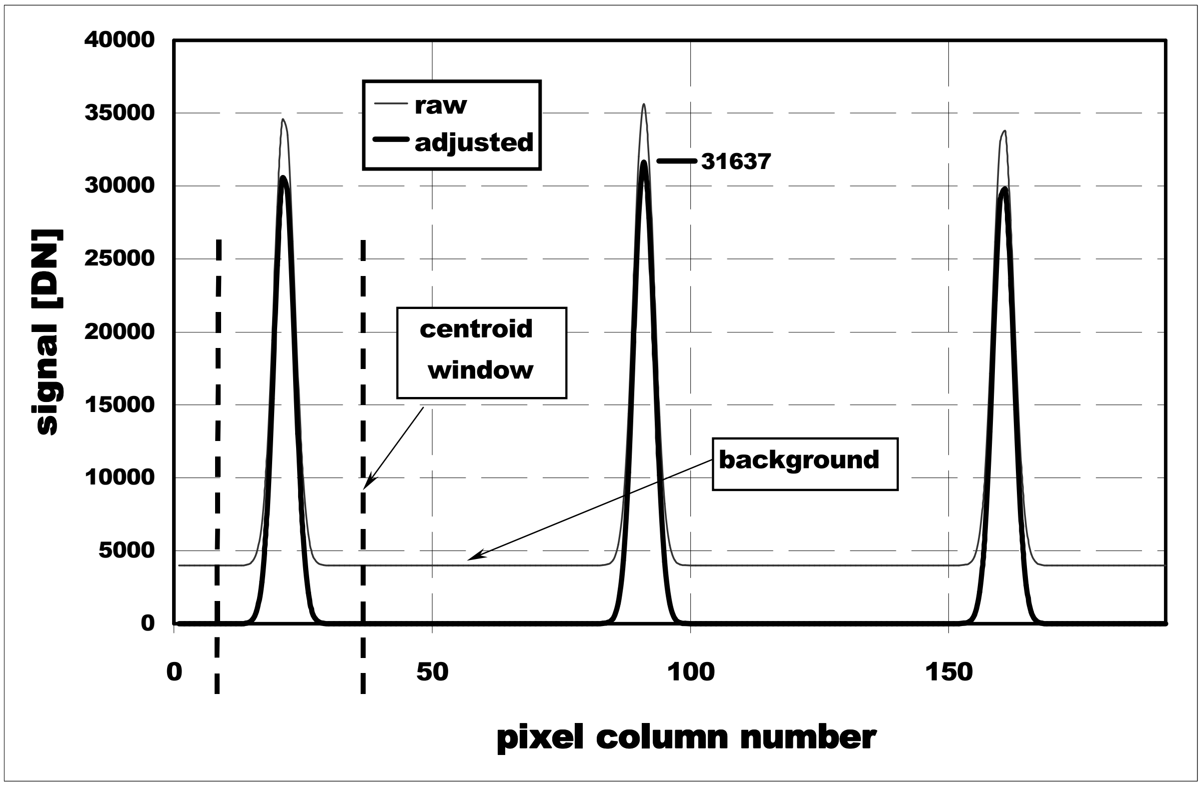


Figure 4 – sequential processing steps of encoder algorithm are illustrated: a) bottom, top, and middle of row markers are found; b) ends of three row markers are detected and lengths of row markers are found to be within range; c) pixels at centers of code bits are located through known row and column offsets, and code groups are deciphered by looking at those pixels' brightnesses with associated binary weights; d) fiducials are found to occupy rows above code bits; centroiding and background windows are defined with respect to ends of row markers, and fiducial centroids are



The centroids of the fiducials are computed within a window nominally given by the ends of the row markers. The column number of the center column of pixels (we will use 96 in our example) is subtracted from each centroid so that centroid values to the left of the center column are negative and those to the right are positive. We find that the centroids of fiducials 1, 2, and 3 are -74.635 pixels, -5.014 pixels and 64.587 pixels, respectively. (These numbers were made up to include typical uncertainties in determining fiducial centroids.)

So far, the algorithm has not needed to know scale pitch. But now, we are ready to determine the encoded position and scale pitch comes into play. So, let us assume that the scale is a sector scale of rotary format and that the angular spacing of adjacent fiducials is 0.1 degrees. We expected that resolution will be around 0.00001 degrees. First we need to know the image scale in degrees per pixel. This scale is just 0.1 degrees divided by the differences in centroid locations of adjacent fiducials. The scale for fiducial 1 is 0.10000 degrees / 69.621 pixels = 0.001436569 degrees per pixel. The scale for fiducial 3 is 0.10000 degrees / 69.601 pixels = 0.001436561 degrees per pixel. The difference in these image scales is actually in the noise. The scale for fiducial 2 is taken to be 0.10000 degrees / 139.222 pixels over two fiducials = 0.001436553 degrees per pixel.

Finally, the algorithm calculates the position of each fiducial according to Eq. 1:

Position $\boldsymbol{r_1}$ = 96 x 0.1 - (-74.635) x 0.001436569 = **9.70722** degrees

Position r_2 = 97 x 0.1 - (- 5.014) x 0.001436553 = **9.70720** degrees

Position r_3 = 98 x 0.1 - (64.587) x 0.001436561 = <u>**9.70722**</u> <u>degrees</u>

average encoded position = **9.70721** degrees

Note that the derived position value from the first digit behind the decimal point leftward has only to do with correct deciphering of code identities and has nothing to do with the algorithm's centroiding aspects. Effectively, position determination amounts to identifying integer cycle values and phase within a cycle. This makes coarse determination of position very fast and fine determination of position very sensitive.

A strong distinction between NASA's pattern recognition encoders and conventional encoders is best made here. The process of centroiding used in NASA's encoders must not be confused with the process of interpolation of quadrature signals — assumed to be truly sinusoidal — by which conventional encoders extend their resolution. While both processes determine phase within a cycle of their scale patterns, centroiding is a linear process whose accuracy is limited by placement of the centers of scale fiducials, any fixed, geometric anamorphicity in scale imaging, and noise. For interpolation in conventional encoders, purity of sinusoidality, phase, and amplitude of quadrature signals (among other things) is paramount to accuracy. These signal aspects depend ultimately on accuracy of scale pattern edge transitions which depend on details and uniformity of scale fabrication — from photo-patterning, through pattern development and scale etching.

For these reasons, most encoder practitioners limit interpolation to about 8 bits (1/256 of a cycle), and cycle errors which occur tend to be only partially deterministic, complicating calibration. In contrast, exact edge positions of scale markings are quite unimportant to NASA's encoders as long as fiducials' center placements are accurate. Since edge locations themselves do not matter, pattern development and scale etching uniformity are not critical for NASA's encoders. For similar encoder resolutions, cycle frequency of a conventional encoder scale, and thus the frequency of cyclic errors, is roughly 40 times higher than for NASA's encoder, whose cyclic errors tend to be very low in amplitude and very deterministic, simplifying calibration.

One final operation adjusts the exposure to optimize signal-to-noise ratio in the next image taken. The algorithm expects the actual peak column value to be within 1000 DN of that which would have occurred if all 132 rows contributing to this peak column value had all been just 5 LSB's shy of digital saturation (at 250 DN). This means that the expected signal should have been between 32,000 and 34,000 DN. The peak column for this image was found to be 31,637 DN. This image just missed the bottom of that ideal window. So, the brightness value used to moderate LED current is incremented for the next exposure which drives the peak column value upwards, hopefully landing it within the target window.

6. REFERENCES

[1] "Recent advances and applications for NASA's new, ultra-high sensitivity, absolute, optical pattern recognition encoders," D.B. Leviton, SPIE **4091-B42**, San Diego, August 2000

APPENDIX B: DSP Functions

Digital Signal Processor

The Digital Signal Processor (DSP) is a processor that is used specifically for signal processing. In this case the signal is image data of the encoder scale pattern from the CMOS image sensor. An algorithm running on the Digital Signal Processor is used to analyze the image data and then calculate the position of the scale pattern with respect to the sensor. The DSP has a unique design in that it has several arithmetic logic units (ALU's) that allow it to perform complex computations to a signal at very high rates of speed. These chips are often used in applications where large amounts of processing have to be done to a signal in real time such as is needed with Fast Fourier Transform (FFT) or digital filtering applications.

The DSP that is used with the current Leviton Encoder is the Texas Instruments TMS320C6211. This DSP was new to the encoder design as of last year when it was implemented over the floating-point TMS320C6711 DSP due to a speed advantage of 167Mhz compared to 150Mhz. The fixed-point processor is able to perform 1333 MIPS (Mega Instructions Per Second, compared to the C6711 that preformed only 900 MFLOPS (Mega Floating point Operations per Second). The C6711 and C6211 are almost identical in terms of their operation; both have the same instruction set and memory interface protocols. These similarities make it possible to easily switch between the DSPs, with only minor changes in code to optimize for the unique processor.

In order to aid in development work, the DSP is included on an interface board such as is seen in Figure 2.8. This setup, known as the Development Starter Kit (DSK), provides a computer interface, additional memory, and simplified connectivity to other devices. There are two primary connectors (J1 and J2) on the DSK for interfacing with peripherals. These two 80 pin connectors include an EMIF (external memory interface) with 32 digital I/O pins, and 22 address pins as well as various hardware interrupts and voltage supply pins. The full pin out of these connectors can be found in the Texas Instruments data sheet for this DSK. The DSK also provides for simple programming of the DSP. System code, such as the image-processing algorithm is written using an included software package called Code Composer. This programming interface is almost identical to C code, however it provides unique functionality for viewing the registers on the DSP as well as controlling internal protocols such as Direct Memory Access (DMA), system timer, and serial ports. This interface also makes it possible to directly modify the assembly code that is generated by the compiler. When compiled, an output file is created that can be loaded onto the

DSK via a parallel interface using the Code Composer software. The Code Composer interface allows for debugging of the DSP program by using software interrupts to step through the code and track the output.

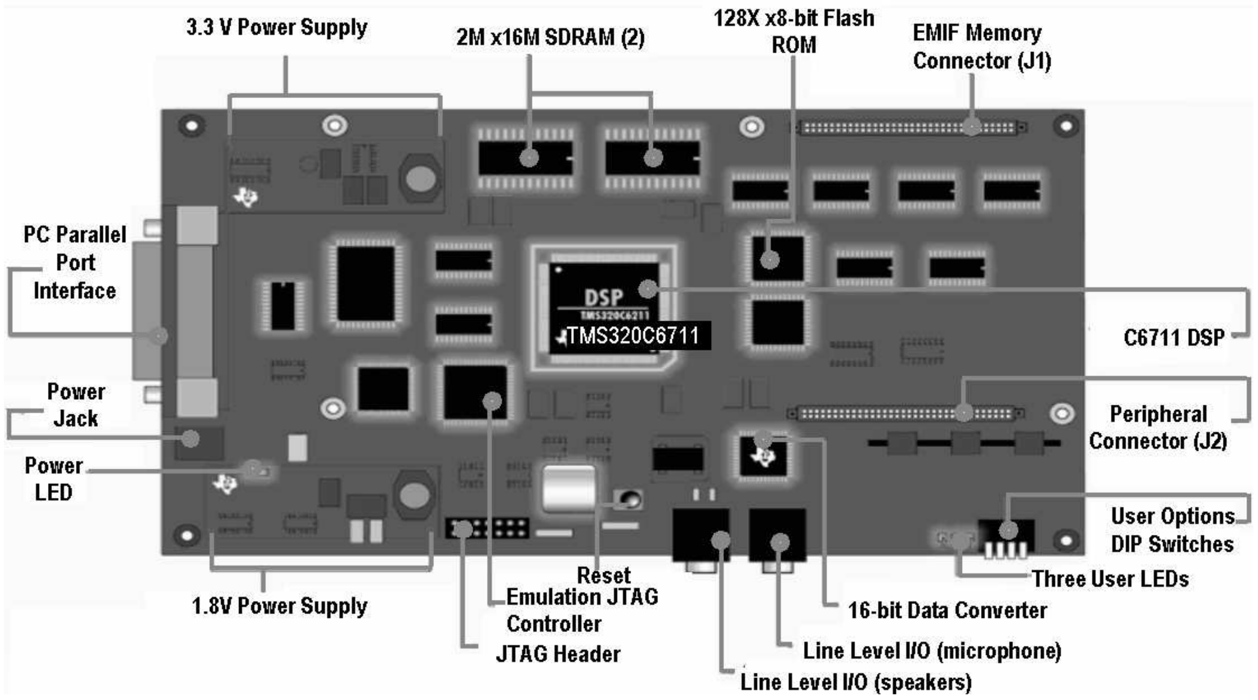

3.3 V Power Supply
2M x16M SDRAM (2)
128X x8-bit Flash ROM
EMIF Memory Connector (J1)
PC Parallel Port Interface
DSP
TMS320C6711
C6711 DSP
Power Jack
Peripheral Connector (J2)
Power LED
User Options DIP Switches
Three User LEDs
Reset
1.8V Power Supply
Emulation JTAG Controller
16-bit Data Converter
JTAG Header
Line Level I/O (microphone)
Line Level I/O (speakers)

Figure 2.8 – Texas Instruments TMS320C6711 DSK Board (Graphic by Texas Instruments).

Enhanced Direct Memory Access (EDMA)

Enhanced Direct Memory Access (EDMA) is a memory transfer protocol optimized for moving data efficiently to and from external devices. The interface has the capability of transferring up to 1500Mb/s using quick direct memory access (QDMA) block transfers (Texas Instruments, 2001). Both the C6211 and C6711 have 16 EDMA channels that can independently and simultaneously receive requests for data transfers. Once an EDMA transfer is requested on a given channel the operation is put in a priority queue and waits to be executed. Registers on the DSP are used to set the priority of a given EDMA channel

A transfer request over an EDMA channel is initialized by a trigger event. A specific trigger, that cannot be changed, is associated with each channel. Channels 4,5,6 and 7 are important because their events are external hardware interrupt pins, meaning that a signal on an external pin triggers the event. When an event is detected on a given channel the appropriate data in the event register is set, immediately beginning the data transfer. There is also an event clear register that can be set to stop the triggering of events from the event register. Information can be transferred over each of the EDMA channels by three different methods, element synchronized, frame synchronized, or QDMA block transfer (Texas Instruments, 2001). The EDMA protocol also allows for address incrementing, meaning that on each trigger event the destination or source address can be modified to store or retrieve information from consecutive memory locations. This is useful if information larger then an array length or element must be moved.

EDMA transfers can optimize the performance of the processor because the transfers are performed independently of digital signal processor operation. This allows the processor to be used for other events while concurrent data transfer is occurring. EDMA is also nearly three times faster then having the processor directly access the external data port.

Multichannel Buffered Serial Port (McBSP)

McBSP, or Multichannel Buffered Serial Port is based on the standard serial port interface. This serial port provides full duplex communication and multichannel transmits and receives of up to 128 channels. All data registers are double buffered to allow for continuous data streaming of 8,12,16,20,24, and 32 bit data transmit sizes. One of the most useful functions of the McBSP is clock signal generation. The McBSP has a sample rate generator that can be programmed by register to produce clock frequency of ½ the core clock speed of the DSP, divided by integers between 1 and 255. Several other registers are available for configuring the 2 McBSP ports that are provided on the J2 connector.

External Memory Interface (EMIF)

EMIF (External Memory Interface) is the standard memory interface for the DSP, which includes 32 standard digital I/O pins and 22 address pins that are designed to interface with external memory such as RAM (Random Access Memory). The EMIF bus is a single input and output bus, which means that anything connected to the bus, will "see" any data that is being transferred between the DSP and other devices. For this reason the address pins are provided to indicate where the data on the data I/O pins is supposed to go. The EMIF interface has several control registers associated with it, allowing EMIF to be configured to interface with specific types of memory. The control space registers control memory interface factors such as setup, strobe, and hold times. This interface also has an output clock pin that can be configured to provide a clock control signal to memory. An input clock pin is also available that allows for an input clock that can be provided to the DSP externally.

The EMIF I/O pins are mapped through Code Composer's GEL file to addresses starting at 0xA0000000 hex. These external pins are capable of addressing up to 256Mb of memory. The mapable memory space for the C6711 DSK can be seen below in Figure 2.9. Other mapped memory also depicted in this diagram includes the internal 64KB of memory on the DSK; this space is available for Code Composer code, as well as immediate program storage. There is also memory mapped to on chip peripherals, this includes addressing locations for the various control registers on the DSP, such as EMIF, EDMA, and McBSP.

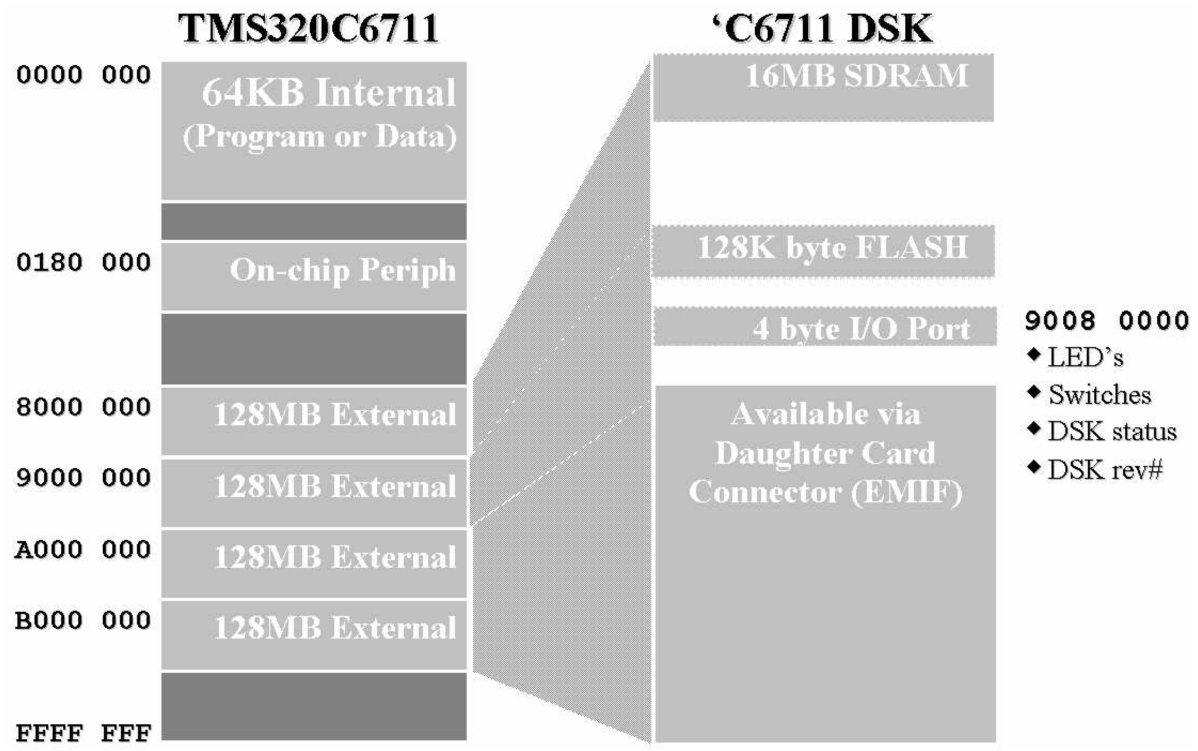


Figure 2.9 – TMSC6711DSK Memory Map (Texas Instruments).

Values on the left are the hex memory addresses where that memory space begins.

APPENDIX C: Sensor Initialization 2004

The following text for sensor initialization is taken from the MQP Report of the 2003 Team. In initializing the sensor, we followed their procedure and obtained exactly the same results.

Digital I/O Register Programming

The first design task that had to be accomplished was to research options for communicating with the control registers on the CMOS APS independent of the Evaluation Kit provided by National Semiconductor. The purpose of this task is to gain a better understanding of the I^2C standard and test that our understanding of the protocol was sufficient before specific register control hardware was developed. To begin this task we had to have an in-depth understanding of the signal that was required to communicate with the CMOS image sensor. Research of the I^2C compatible serial interface was completed and is described in detail in the Background Chapter of this report. With the requirements of the I^2C interface in mind, we studied the output possibilities that could be achieved using an ISA digital input/output card in a PC. The digital I/O card has three different 8-bit ports that can be configured independently for either input or output. This capability was sufficient for sending the signals that were required to program the registers on the CMOS image sensor. This card was used in preliminary development as it could be used to generate the signals that we needed yet did not require any difficult hardware design.

We used several port configurations for the Digital I/O card in testing but the final version used only a single 8-bit port. To control the operation of the Digital I/O card we wrote a Visual Basic program that reads the required clock and data signal information from a text file and outputs the correct binary sequence to the ports of the digital I/O card. The text file containing the signal data was developed in an interactive Excel spreadsheet. The spreadsheet allows the user to input the values for each register and then converts these into the proper data stream. The data stream is saved as a text file that can be read in by the Visual Basic program. The program initialized the I/O card and fed the data stream to the digital output port at the appropriate time. An example of a portion of a data stream output by the Digital I/O card can be seen in Figure 5.2.

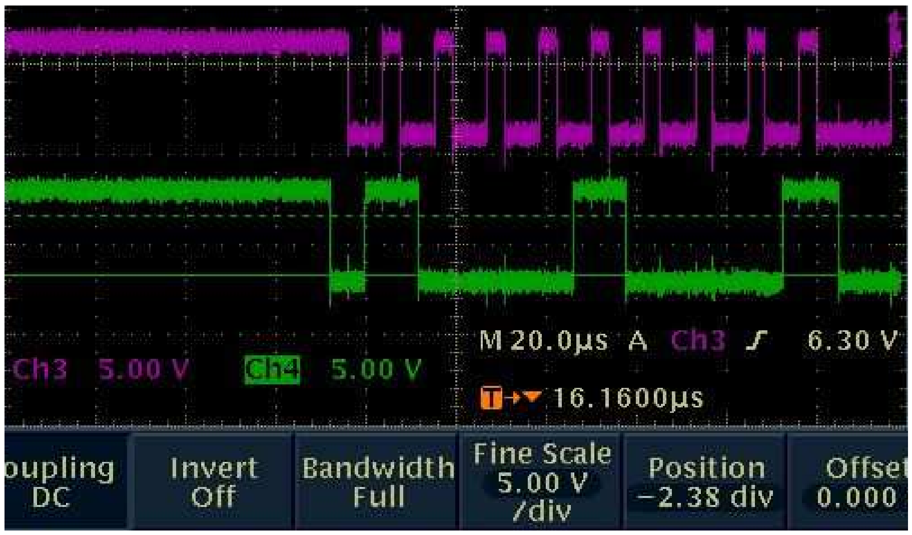
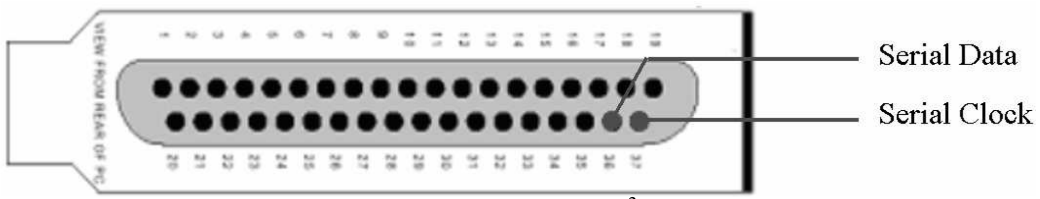


Figure 5.2 – Serial Data and Clock Signals generated by the digital I/O card. The upper channel is the clock signal and the lower channel is the data signal.

Initially we used the two LSBs (least significant bits) on a single output port to create the bit streams. An additional version of the program was written that used two independent ports and allowed the PC to read in the acknowledge signal from the sensor by switching the Digital I/O port to an input. Later we found that this extra step was not required for proper operation. The code required to switch the port between input and output was many times longer (and therefore slower) than the simple code used earlier so that version code was not used in the final version of the program. A diagram of the 2 pins used on the single output port of the Digital I/O card can be seen below in Figure 5.3.



5.3 – Output pins used on the Digital I/O card to create the I^{2}C control signals (Connector Graphic by Measurement Computing Corporation).

Since we were using a single output port in our final configuration, the data values ranged from zero to three. The LSB was attached to the clock line and the 2nd LSB was attached to the data line. The output then had four distinct possibilities, which are shown in Table 5.1. With these possibilities we could generate a data string that would create the proper outputs to allow for register programming. A value was sent at any point the data or clock lines changed, allowing the entire data stream to be represented. In this manner each bit that was clocked out required a sequence of three data output values. Figure 5.4 shows an example of a sample byte clocked to the CMOS image sensor, decimal output to the digital I/O card port is shown as well as the corresponding bit received by the sensor.

Data Output (to port)	Clock line	Data line
0	0	0
1	1	0
2	0	1
3	1	1

Table 5.1 – Data sent to Digital I/O card for a given data line and clock line value.

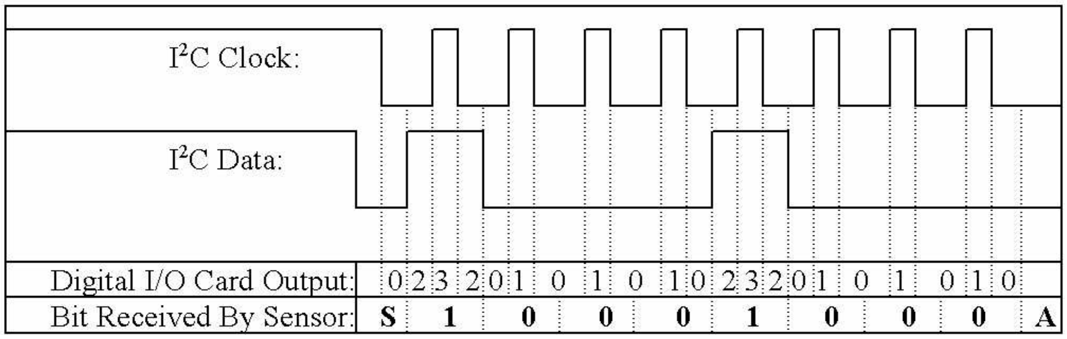

I²C Clock:
I²C Data:
Digital I/O Card Output: 0 2 3 2 0 1 0 1 0 1 0 2 3 2 0 1 0 1 0 1 0
Bit Received By Sensor: S 1 0 0 0 1 0 0 0 A

Figure 5.4 – Digital I/O Card Output, signal on top is the serial clock, signal on the bottom is the serial data.

An initial concern was that the output speed of the input/output card would be too high. However, after conducting several tests in the lab, the 400 kHz maximum input speed of the sensor turned out to be several times the maximum output speed of the card, thus invalidating the concern (National Semiconductor).

Our first version of the program created a pair of bit streams that appeared correct when displayed on an oscilloscope, however the signals failed to program the registers on the sensor. We were concerned that the Digital I/O card was supplying dangerously high currents to the CMOS image sensor during the acknowledge phase when the sensor tries to ground the high signal sent by the card. In an attempt to correct for this, our next version of code allowed the data pin to float after the data stream was sent so that the acknowledge signal would not cause a short in the system. This version of the program was successful at programming a single register, however the code was inconsistent and unreliable, particularly when attempting to program multiple registers.

At this point we figured that there was something critical that we were missing which was preventing the sensor from accepting the programming every time. We discovered that the output from the card is a 5-volt TTL signal and the sensor requires a 3.3-volt input. On occasion the sensor would accept the 5-volt signal, but after we setup a simple resistor voltage divider that reduced the signal to 3.3 volts the sensor accepted the programming every time. As a result of the voltage divider configuration that was used to provide the 3.3V signal, the current was also limited, allowing us to operate the card in output mode all the time as the sensor had sufficient ability to sink the smaller current source, without being damaged. With the voltage mismatch problem solved we resorted to an earlier version of the program and used a single output port to create the bit streams.

Register Programming With DSP

The task to be accomplished in the next stage of development was to use the DSP to program the registers on the CMOS active pixel sensor. To program the registers using the DSP, Code Composer was used to develop a C program that would send out values 0,1,2, or 3 (in binary) to external pins on the DSK similar to the method used with the Digital I/O card and FPGA. These two binary bits represent the clock and data signals of the I^2C interface, which are needed to program the CMOS image sensor.

In the first version of the C code developed for register programming, data was output over the EMIF (external memory interface) bus, which starts at the mapped address 0xA0000000. In the register control program each of the register values were hard coded into an array of unsigned characters. Several functions were developed that generate the appropriate values on the EMIF bus for start, end, and acknowledge conditions, as well as to increment through these arrays and output the correct values to the EMIF pins defined by the address in the header file.

This version of the DSP register programmer used an addressing process to eliminate the potential of other data sent over the EMIF I/O pins affecting the operation of the CMOS image sensor. Each time data is sent to the EMIF I/O pins the corresponding address that is defined in the header file is sent to the EMIF address pins. This address is sent along with a data pin through an AND gate; another AND gate is used for the other bit. The output of these gates is sent to the corresponding I^2C control pins on the CMOS image sensor. This process of ANDing the data with the address means that the output of the AND gate will be zero unless the correct address has been enabled, at which point the data will appear on the output of the AND gate which is connected to the sensor. One disadvantage of this gating process is that the EMIF address pins are not latched. This means that when data is sent to the data pins at a given address, the address on the address pins is not held, only pulsed on the address bus. For this reason the data has to be sent to the data pins repeatedly, so that the address is held high sufficiently long for the sensor to recognize the data. To accomplish this each function in this version of the register programmer has a loop that sends the data 25 times each time a value is sent to the port.

The process of having to send data to the EMIF I/O pin multiple times is inefficient. To eliminate the need for addressing, research was done on the DSK to determine other output pins that are available. We found that address 0x90080000 connects to 8 bits of digital I/O pins. The program was modified to send the clock and data output to these pins CNTL1 (serial clock) and CNTL0 (serial data). This modification allowed for the removal of the loops that were required to send the data multiple times. The removal of addressing from the system also allowed for the removal of the AND gates and support logic that were previously required. With this program structure the sensor could be programmed by connecting directly to serial clock and data pins on the DSK. This was the final method used for programming the registers on the image sensor.

APPENDIX D: Final 2004 Code

The VHDL and Visual Basic codes are provided in the CD.

DSP Image-Processing Code:

```
#include "ControlReg.h"
#include "registerspgrm.c"

#include <csl_edma.h>
#include <csl_emif.h>
#include <csl.h>
#include <stdio.h>
#include <csl_mcbsp.h>
#include "registerspgrm.h"

//Initializing Global Variables

unsigned int image[1440];                    // allocating space for image
unsigned int default_io;
unsigned int Fthresh;                        // Fiducial Threshold
unsigned int Cthresh = 0x00034000;   // Code Bit Threshold
unsigned int Bias;                           //Background Black level Bias
double PosBuff[500];
int ColumnNumb[4];
double CNUM[4];                                     //Centroid Numerator
double CDEN[4];                                     //Centroid Denominator
double centroid[4];
double DegreePerPix[4];
unsigned int  Codebits[4];
double position[4];
double FinalPOS = 0.0;              //Position
double imagectr = 239.5;            //Center of Window
double scale = 580E-6;        //Scale pitch

int *CE2CTL_ADDR_ptr = (int *)CE2CTL_ADDR;
int *EXTPOL_ADDR_ptr  = (int *)EXTPOL_ADDR;

void CONFIG_IO();
void CONFIG_Edma();
void WAIT_Data();
void Mask_Data();
void FindCol();
void GetBias();
void ComputeCentroid();
void ComputeCodebits();
void ComputePosition();




EDMA_Handle hEdmaExtint4;


EDMA_Config EDMACfg0 = {
        0x213A0001,           // OPT Register - 32bit wide, sum=dum=1, using interrupt 10
        0xA0000000,           // source address- EMIF<- Dual port Memory of FPGA
        0x000005A0,           // transfer count- 1440 values or 5A0 Hex
        (unsigned int)&image[0],              // Internal Memory starts at image[0]
        0x00000000,           // Index update parameter              (index)
        0x000105A0            // Element count Reload and Link Address (7E0 is Null address)
};



int main()
{
int i;
FILE *fp;
fp = fopen("DATA2", "a");

CSL_init();             //initilize CSL functionality
//intsen();                     //Initilize sensor
*registers = 0x00000000;  // clear EMIF
Delay(1000);
CONFIG_IO();            //Config EMIF for Asynchronous block transfer
CONFIG_Edma();          // Set up EDMA block transfer

//for (i = 0; i <=500; i++)
//{
WAIT_Data();                    // Wait for data to be transferred
Mask_Data();                    // Mask out high 12 bits, remove garbage data
FindCol();                      // Find 4 Column high points
GetBias();                      // Compute background Black level
ComputeCentroid();   // Compute the 4 Centroids and Degree per Pixel
ComputeCodebits();  // Compute Codebit Values
ComputePosition();  // Compute Exact Position

//PosBuff[i] = FinalPOS;
fprintf(fp,"%e\n", FinalPOS);
//}
```

```
//printf("DataBuff =%p\n",&PosBuff[0]);
printf("FinalPOS =%e\n",FinalPOS);
EDMA_disableChannel(hEdmaExtint4);  // closing EDMA channel
EDMA_close(hEdmaExtint4);
return 0;
}


void CONFIG_IO()
{
        /***************************
        Max Read hold, strobe, and setup
        Write values don't matter
        ****************************/
        default_io = 0x000F3F27;
        *CE2CTL_ADDR_ptr = default_io;

        return;

}


void CONFIG_Edma()
{

        *EXTPOL_ADDR_ptr = 0x00000000;                  // making interupts active on low to
high trans
        hEdmaExtint4 = EDMA_open(EDMA_CHA_EXTINT4,EDMA_OPEN_RESET);
        EDMA_config(hEdmaExtint4, &EDMACfg0);
        EDMA_enableChannel(hEdmaExtint4);  // opening EDMA channel
        return;

}

void WAIT_Data()
{

        EDMA_RSET(CIPR,(1<<10));
        while (!((Uint32)EDMA_RGET(CIPR) & (1 << 10)));  //Waiting until transfer completes
        return;
}

void Mask_Data()
{
        int i;
        for (i=0; i<=1439; i++)
        {
                image[i] &= 0x000FFFFF; // Masks out top 12 unused bits
                }
        return;
}

void FindCol()
{
        int x;
        int i;
        int y;
        int Max = 0;
        int prval = 0;
        for (i= 0; i <=3; i++)
        {
        ColumnNumb[i] = 0;
        }
        Fthresh = 0;

        //computing threshold 3/4 the max value
        for (i = 0; i <= 1439; i++)
        {
                if (image[i] >= image[Max])
                {
                        Max = i;
                }
        }

        Fthresh = ((image[Max]*3)/4);

        // Finding 4 Fiducial column values for centroiding
        for (x=109; x <= 301; x+=3)
        {
                if (image[x] > Fthresh)
                {
                        if (image[x] > image[prval])
                                        prval = x;
                        else
                                {
                                        ColumnNumb[0] = prval;
                                        break;
                                 }
                }
        }
        prval = ColumnNumb[0] + 204;
```

```
	for (y = (ColumnNumb[0] + 204); y <= (ColumnNumb[0] +300); y+=3)
	{
		if (image[y] > Fthresh)
		{
			if (image[y] > image[prval])
					prval = y;
			else
				{ColumnNumb[1] = prval;
					break;
				 }
		 }
	 }
	prval = ColumnNumb[1] + 204;
	for (y = (ColumnNumb[1] + 204); y <= (ColumnNumb[1] +300); y+=3)
	{
		if (image[y] > Fthresh)
		{
			if (image[y] > image[prval])
					prval = y;
			else
				{ColumnNumb[2] = prval;
					break;
				 }
		 }
	 }
	prval = ColumnNumb[2] + 204;
	for (y = (ColumnNumb[2] + 204); y <= (ColumnNumb[2] +300); y+=3)
	{
		if (image[y] > Fthresh)
		{
			if (image[y] > image[prval])
					prval = y;
			else
				{ColumnNumb[3] = prval;
					break;
				 }
		 }
	 }
	return;
}
void GetBias()
{
	// Averaging 17 black values to get Background Bias
	int Dark = 0;
	int x;
	int Total = 0;
```

```
          Dark = ((ColumnNumb[1] - ColumnNumb[0])/2)+ColumnNumb[0];
          for (x = Dark-24; x <= Dark+24; x+=3)
                    {
                     Total += image[x];
                    }
          Bias = (Total/17);
return;
}

void ComputeCentroid()
{
          /****************************************************\
          Computing centroids using a window of 30 from peak fiducial
          Value.  Also calculating degrees per pixel to ensure proper
          units
          \*****************************************************/

          int i;
          int x;
          int m;
          for (m = 0; m <=3; m++)
          {
          CNUM[m] = 0.0;
          CDEN[m] = 0.0;
          centroid[m] = 0.0;
          DegreePerPix[m] = 0.0;
          }

          for (i = 0; i<=3; i++)
          {
                    for (x = (ColumnNumb[i]-45); x <= (ColumnNumb[i]+45); x+=3)
                    {
                              CNUM[i] += ((image[x]-Bias)*((x-1)/3));
                              CDEN[i] += (image[x]-Bias);
                    }
                    centroid[i] = ((CNUM[i]/CDEN[i])-imagectr);
          }

                    DegreePerPix[0] = (scale/(centroid[1]-centroid[0]));
                    DegreePerPix[1] = (scale/(centroid[2]-centroid[0]))*2;
                    DegreePerPix[2] = (scale/(centroid[3]-centroid[1]))*2;
                    DegreePerPix[3] = (scale/(centroid[3]-centroid[2]));
return;
}

void ComputeCodebits()
```

```
{
        /***********************************************\
        Computing Codebit values.  uses a window of 18 for each.
        \***********************************************/

        int x;
        int i;
        unsigned int Zero, One, Two, Three;
        unsigned int Four, Five, Six, Seven;

        for( i = 0; i<=3; i++)
        {
         Codebits[i] = 0x00000000;
          Four = 0;
          for (x= ((ColumnNumb[i]+1)-108); x <= ((ColumnNumb[i]+1)-54); x+=3)
                {
                        Four += image[x];
                }

                if (Four > Cthresh)
                {
                  Codebits[i] += 0x00000010;
                }
          Five =0;
          for (x= ((ColumnNumb[i]+1)-54); x <= ((ColumnNumb[i]+1)); x+=3)
                {
                        Five += image[x];
                }
          if (Five > Cthresh)
          {
                Codebits[i] += 0x00000020;
         }
          Six =0;
          for (x= ((ColumnNumb[i]+1)); x <= ((ColumnNumb[i]+1)+54); x+=3)
                {
                        Six += image[x];
                }
          if (Six > Cthresh)
          {
                Codebits[i] += 0x00000040;
          }
          Seven =0;
          for (x= ((ColumnNumb[i]+1)+54); x <= ((ColumnNumb[i]+1)+108); x+=3)
                {
                        Seven += image[x];
                }
```

```
            if (Seven > Cthresh)
            {
                    Codebits[i] += 0x00000080;
                    }
            Zero = 0;
            for (x= ((ColumnNumb[i]+2)-108); x <= ((ColumnNumb[i]+2)-54); x+=3)
                    {
                            Zero += image[x];
                    }
            if (Zero > Cthresh)
            {
                    Codebits[i] += 0x00000001;
            }
            One = 0;
            for (x= ((ColumnNumb[i]+2)-54); x <= ((ColumnNumb[i]+2)); x+=3)
                    {
                            One += image[x];
                    }
            if (One > Cthresh)
            {
                    Codebits[i] += 0x00000002;
            }
            Two = 0;

            for (x= ((ColumnNumb[i]+2)); x <= ((ColumnNumb[i]+2)+54); x+=3)
                    {
                            Two += image[x];
                    }
            if (Two > Cthresh)
            {
                    Codebits[i] += 0x00000004;
            }
            Three = 0;
            for (x= ((ColumnNumb[i]+2)+54); x <= ((ColumnNumb[i]+2)+108); x+=3)
                    {
                            Three += image[x];
                    }
            if (Three > Cthresh)
            {
                    Codebits[i] += 0x00000008;
                    }
    }
    /*
            printf("Codebits0 = %i\n", Codebits[0]);
            printf("Codebits1 = %i\n", Codebits[1]);
            printf("Codebits2 = %i\n", Codebits[2]);
```

```
	printf("Codebits3 = %i\n", Codebits[3]);
*/
return;
}
void ComputePosition()
{
	/*******************************************************\
	Computing final position.  Each position is averaged to find the
	Final position.
	\*******************************************************/
	position[0] = (Codebits[0]*scale) - (centroid[0]* DegreePerPix[0]);
	position[1] = (Codebits[1]*scale) - (centroid[1]* DegreePerPix[1]);
	position[2] = (Codebits[2]*scale) - (centroid[2]* DegreePerPix[2]);
	position[3] = (Codebits[3]*scale) - (centroid[3]* DegreePerPix[3]);

	FinalPOS = ((position[0] + position[1] + position[2] + position[3])/4);


	return;

}
```

DSP Code for Sensor Initialization

```
#include <stdio.h>
#include <time.h>
#include "registerspgrm.h"

int *output = (int *)OUTPUT;
int *registers = (int *)REGISTERS;

//Function Definitions
void Delay(int max);   //Creates a delay in proportion to the passed value
void StartCondition(); //Sends a "Start" condition to the sensor
void StopCondition(); //Sends a "Stop" condition to the sensor
void Acknowledge();		//Sends an "ACK" to the sensor
void outputconv(unsigned char dat);  //Outputs the appropriate sequence for the input data
void OutputRegister(unsigned char deviceadd[8], unsigned char regadd[8], unsigned char
regdata[8]); //Outputs a complete sequence to a register on the sensor
void intsen();


void intsen()
{
```

```
unsigned char deviceid[8] = {1,0,1,0,1,0,1,0};  //Loads the Device ID into an array
//This section loads the device register IDs and values
//Device ID and Revision

//Configuration and Power Control
unsigned char r05[8] = {0,0,0,0,0,1,0,1}, d05[8] = {0,0,0,0,0,0,0,0}; //05h = vclkgen = 00h
unsigned char r06[8] = {0,0,0,0,0,1,1,0}, d06[8] = {0,0,0,0,0,0,0,0}; //06h = pwdnrst = 00h
unsigned char r07[8] = {0,0,0,0,0,1,1,1}, d07[8] = {1,0,1,0,1,0,1,0}; //07h = I2Cmode = AAh

//video mode
unsigned char r09[8] = {0,0,0,0,1,0,0,1}, d09[8] = {0,0,0,0,0,1,1,1}; //09h = Optctrl = 07h

//Video and Scan
unsigned char r10[8] = {0,0,0,1,0,0,0,0}, d10[8] = {0,0,0,0,0,0,0,0}; //10h = Vidconfig = 01h
unsigned char r11[8] = {0,0,0,1,0,0,0,1}, d11[8] = {0,0,0,0,0,0,0,0}; //11h = Vscan = 00h
unsigned char r13[8] = {0,0,0,1,0,0,1,1}, d13[8] = {0,0,0,0,0,1,0,0}; //13h = Hscan = 04h
unsigned char r15[8] = {0,0,0,1,0,1,0,1}, d15[8] = {0,0,0,0,0,0,0,0}; //15h = Itimeconfig = 00h

//Windowing Function
unsigned char r19[8] = {0,0,0,1,1,0,0,1}, d19[8] = {0,0,0,0,0,0,0,1}; //19h = Wrows = 01h
unsigned char r1A[8] = {0,0,0,1,1,0,1,0}, d1A[8] = {0,0,1,1,1,1,0,0}; //1Ah = Wrowe = 3Ch
unsigned char r1B[8] = {0,0,0,1,1,0,1,1}, d1B[8] = {0,0,0,0,0,1,1,1}; //1Bh = Wrowlsb = 07h
unsigned char r1C[8] = {0,0,0,1,1,1,0,0}, d1C[8] = {0,0,0,0,0,0,0,1}; //1Ch = Wcols = 01h
unsigned char r1D[8] = {0,0,0,1,1,1,0,1}, d1D[8] = {0,1,0,1,0,0,0,0}; //1Dh = Wcole = 50h
unsigned char r1E[8] = {0,0,0,1,1,1,1,0}, d1E[8] = {0,0,0,0,0,1,1,1}; //1Eh = Wcollsb = 07h

//Frame Rate
unsigned char r20[8] = {0,0,1,0,0,0,0,0}, d20[8] = {0,0,0,0,0,0,0,0}; //20h = Fdelayh = 00h
unsigned char r21[8] = {0,0,1,0,0,0,0,1}, d21[8] = {0,0,0,0,0,0,0,0}; //21h = Fdelayl = 00h
unsigned char r22[8] = {0,0,1,0,0,0,1,0}, d22[8] = {0,0,0,0,0,0,0,0}; //22h = Rdelayh = 00h
unsigned char r23[8] = {0,0,1,0,0,0,1,1}, d23[8] = {0,0,0,0,1,0,0,0}; //23h = Rdelayl = 08h
unsigned char r24[8] = {0,0,1,0,0,1,0,0}, d24[8] = {0,0,0,0,0,0,0,0}; //24h = Itimeh = 00h
unsigned char r25[8] = {0,0,1,0,0,1,0,1}, d25[8] = {0,0,0,0,0,0,0,0}; //25h = Itimel = 00h

//Black Level
unsigned char r40[8] = {0,1,0,0,0,0,0,0}, d40[8] = {0,0,0,0,1,0,0,0}; //40h = Blklevconfig = 08h
unsigned char r41[8] = {0,1,0,0,0,0,0,1}, d41[8] = {0,0,0,0,1,0,1,0}; //41h = Blktarget = 10h

//Programmable Gain Channel
unsigned char r42[8] = {0,1,0,0,0,0,1,0}, d42[8] = {0,0,0,0,1,1,1,1}; //42h = PGA = 0Fh

//Offset
unsigned char r46[8] = {0,1,0,0,0,1,1,0}, d46[8] = {0,0,0,0,0,0,0,0}; //46h = Offset = 00h
unsigned char r4A[8] = {0,1,0,0,1,0,1,0}, d4A[8] = {0,0,0,0,0,0,0,0}; //4Ah = Reserved register,
set to 0h
```

```
//Adjustment
unsigned char r50[8] = {0,1,0,1,0,0,0,0}, d50[8] = {0,0,0,0,1,0,0,0}; //50h = Vsyncadust = 08h
unsigned char r51[8] = {0,1,0,1,0,0,0,1}, d51[8] = {0,0,0,0,1,0,0,0}; //51h = Hsynadjust = 08h
unsigned char r52[8] = {0,1,0,1,0,0,1,0}, d52[8] = {0,0,0,0,0,0,0,0}; //52h = Dvbusconfig0 = 00h
unsigned char r53[8] = {0,1,0,1,0,0,1,1}, d53[8] = {0,0,0,0,1,1,1,1}; //53h = Dvbusconfig1 = 0Fh
unsigned char r54[8] = {0,1,0,1,0,1,0,0}, d54[8] = {1,0,1,0,0,0,0,0}; //54h = Dvbusconfig2 =
A0h
unsigned char r55[8] = {0,1,0,1,0,1,0,1}, d55[8] = {0,0,0,0,0,0,0,0}; //55h = Dvbusconfig3 = 00h

//Factory Test Registers
unsigned char r80[8] = {1,0,0,0,0,0,0,0}, d80[8] = {0,0,0,0,0,0,0,0}; //80h = intreg1 = 00h
unsigned char r83[8] = {1,0,0,0,0,0,1,1}, d83[8] = {0,1,0,1,1,1,1,0}; //83h = Pixeloffset = 5Eh
unsigned char r85[8] = {1,0,0,0,0,1,0,1}, d85[8] = {1,0,0,0,0,0,1,0}; //85h = Pwctrl = 82h
unsigned char r88[8] = {1,0,0,0,1,0,0,0}, d88[8] = {0,0,0,0,0,0,0,1}; //88h = Intreg2 = 01h

//alternate values
unsigned char r85a[8] = {1,0,0,0,0,1,0,1}, d85a[8] = {1,0,0,0,0,1,1,0}; //85h = Pwctrl = 86h
unsigned char r88a[8] = {1,0,0,0,1,0,0,0}, d88a[8] = {0,0,0,0,0,0,0,0}; //88h = Intreg2 = 00h

/* Initialization Sequence
REGISTERS MUST BE INITIALIZED IN THIS ORDER!!
ALL OTHER REGISTERS CAN BE PROGRAMMED AFTER!!!
*/

OutputRegister(deviceid, r09, d09);
OutputRegister(deviceid, r05, d05);
OutputRegister(deviceid, r53, d53);
OutputRegister(deviceid, r19, d19);
OutputRegister(deviceid, r1A, d1A);
OutputRegister(deviceid, r1B, d1B);
OutputRegister(deviceid, r1C, d1C);
OutputRegister(deviceid, r1D, d1D);
OutputRegister(deviceid, r1E, d1E);
OutputRegister(deviceid, r53, d53);
OutputRegister(deviceid, r88, d88);
OutputRegister(deviceid, r85, d85);
Delay(1000);
OutputRegister(deviceid, r85a, d85a);
OutputRegister(deviceid, r88a, d88a);
OutputRegister(deviceid, r4A, d4A);
OutputRegister(deviceid, r40, d40);

//ALL OTHER REGISTERS CAN BE PROGRAMMED NOW

OutputRegister(deviceid, r20, d20);
OutputRegister(deviceid, r21, d21);
OutputRegister(deviceid, r22, d22);
OutputRegister(deviceid, r23, d23);
OutputRegister(deviceid, r24, d24);
OutputRegister(deviceid, r25, d25);
OutputRegister(deviceid, r11, d11);
OutputRegister(deviceid, r41, d41);
OutputRegister(deviceid, r42, d42);
OutputRegister(deviceid, r46, d46);
OutputRegister(deviceid, r50, d50);
OutputRegister(deviceid, r51, d51);
OutputRegister(deviceid, r52, d52);
OutputRegister(deviceid, r54, d54);
OutputRegister(deviceid, r55, d55);
OutputRegister(deviceid, r83, d83);

*registers = 0x00000000;

/*
OutputRegister(deviceid, r06, d06);
OutputRegister(deviceid, r07, d07);
OutputRegister(deviceid, r10, d10);
OutputRegister(deviceid, r13, d13);
OutputRegister(deviceid, r15, d15);
OutputRegister(deviceid, r40, d40);
OutputRegister(deviceid, r53, d53);
*/

}

void Delay(int max)    //Causes a delay for "max" cycles
                       {
        int del = 0;
        int u=0;
        while (del != max)
        {
        while (u != 1000)
        {
        u++;
        }
        del++;
        }
        }
void StartCondition()  //Sends a "Start" condition to the sensor
        {
        *registers =0x18000000;
        Delay(1);
        *registers =0x10000000;
        Delay(1);
        *registers =0x00000000;
        Delay(1);
        }
void StopCondition()  //Sends a "Stop" condition to the sensor
        {
        *registers =0x00000000;
        Delay(1);
        *registers =0x10000000;
        Delay(1);
        *registers =0x18000000;
        Delay(1);
        }
void Acknowledge()   //Sends a "Acknowledge" sequence to the sensor
        {
        *registers =0x00000000;
        Delay(1);
        *registers =0x10000000;
        Delay(1);
        *registers =0x00000000;
        Delay(1);
        }
void outputconv(unsigned char dat)   //Outputs the appropriate sequence for each bit read in
        {
        if (dat == 0)
                {
                *registers =0x00000000;
                Delay(1000);
                *registers =0x10000000;
                Delay(1000);
                *registers =0x00000000;
                Delay(1000);
                }
        else
                {
                *registers =0x08000000;
                Delay(1000);
                *registers =0x18000000;
                Delay(1000);
                *registers =0x08000000;
                Delay(1000);
                }
        }
void OutputRegister(unsigned char deviceadd[8], unsigned char regadd[8], unsigned char
regdata[8])       //Outputs the corrrect data stream to program a register (values are passed in)
```

```
        {
        unsigned char temp;
        unsigned char temp2;
        StartCondition();
        temp = 0;
        while (temp != 8)
                {
                temp2 = deviceadd[temp];
                outputconv(temp2);
                temp++;
                }
        Acknowledge();
        temp = 0;
        while (temp != 8)
                {
                temp2 = regadd[temp];
                outputconv(temp2);
                temp++;
                }
        Acknowledge();
        temp = 0;
        while (temp != 8)
                {
                temp2 = regdata[temp];
                outputconv(temp2);
                temp++;
                }
        Acknowledge();
        StopCondition();
        }
```

Header File for the Sensor Register File:

```
/* Register Locations
*  Referances:
*  TI SPRU 190D Peripherals Referance Guide
*  TI SPRU 401D      Chip Support Library API User's Guide
*/
#ifndef ControlReg_H
#define ControlReg_H

//EDMA Control Register
// Define Channel Interupt and Sevicing Control Regesters
#define PQSR_ADDR                                   0x01A0FFE0  // Priority queue status reg
#define CIPR_ADDR                           0x01A0FFE4  // Channel interrupt pending reg
```

```
#define CIER_ADDR                    0x01A0FFE8  // Channel interrupt enable reg
#define CCER_ADDR                              0x01A0FFEC  // Channel chain enable reg
#define ER_ADDR                                0x01A0FFF0  // Event reg
#define EER_ADDR                     0x01A0FFF4  // Event enable reg
#define ECR_ADDR                     0x01A0FFF8  // Event clear reg
#define ESR_ADDR                     0x01A0FFFC  // Event set reg

// Define QDMA channel  Register addresses
#define QDMA_OPTIONS_ADDR                              0x02000000 // QDMA, options
#define QDMA_SRC_ADDR_ADDR                   0x02000004 // QDMA, SRC
#define QDMA_CNT_ADDR                                  0x02000008 // QDMA, array/frame
count| QDMA, elment count
#define QDMA_DST_ADDR_ADDR                                       0x0200000C // QDMA, DST
#define QDMA_IDX_ADDR                                  0x02000010 // QDMA, array/frame
index| QDMA, elment index

// Define QDMA channel Psuedo Register addresses
#define QDMA_S_OPTIONS_ADDR                            0x02000020 // QDMA, options */
#define QDMA_S_SRC_ADDR_ADDR                 0x02000024 // QDMA, SRC */
#define QDMA_S_CNT_ADDR                                0x02000028 // QDMA, array/frame
count| QDMA, elment count */
#define QDMA_S_DST_ADDR_ADDR                           0x0200002C // QDMA, DST */
#define QDMA_S_IDX_ADDR                                          0x02000030 // QDMA,
array/frame index| QDMA, elment index */

// Define EDMA channel addresses (15 channnels w/ 6 registers each)
#define EDMA0_OPTIONS_ADDR                             0x01A00000 /* Event 0, options */
#define EDMA0_SRC_ADDR_ADDR             0x01A00004 /* Event 0, SRC */
#define EDMA0_ARRYFRAM_ELCNT_ADDR       0x01A00008 /* Event 0, array/frame
count| Event 0, elment count */
#define EDMA0_DST_ADDR_ADDR                                      0x01A0000C /* Event 0,
DST */
#define EDMA0_ARRYFRAM_ELIDX_ADDR         0x01A00010 /* Event 0, array/frame
index| Event 0, elment index */
#define EDMA0_ELCNTRELD_LINKADDR_ADDR       0x01A00014 /* Event 0, element
count reload| Event 0, link address */

#define EDMA1_OPTIONS_ADDR                             0x01A00018 /* Event 1, options */
#define EDMA1_SRC_ADDR_ADDR             0x01A0001C /* Event 1, SRC */
#define EDMA1_ARRYFRAM_ELCNT_ADDR       0x01A00020 /* Event 1, array/frame
count| Event 1, elment count */
#define EDMA1_DST_ADDR_ADDR                                      0x01A00024 /* Event 1, DST
*/
#define EDMA1_ARRYFRAM_ELIDX_ADDR         0x01A00028 /* Event 1, array/frame
index| Event 1, elment index */
```

```
#define EDMA1_ELCNTRELD_LINKADDR_ADDR       0x01A0002C /* Event 1, element
count reload| Event 1, link address */

#define EDMA2_OPTIONS_ADDR                          0x01A00030 /* Event 2, options */
#define EDMA2_SRC_ADDR_ADDR                         0x01A00034 /* Event 2, SRC */
#define EDMA2_ARRYFRAM_ELCNT_ADDR       0x01A00038 /* Event 2, array/frame
count| Event 2, elment count */
#define EDMA2_DST_ADDR_ADDR                                   0x01A0003C /* Event 2,
DST */
#define EDMA2_ARRYFRAM_ELIDX_ADDR         0x01A00040 /* Event 2, array/frame
index| Event 2, elment index */
#define EDMA2_ELCNTRELD_LINKADDR_ADDR       0x01A00044 /* Event 2, element
count reload| Event 2, link address */

#define EDMA3_OPTIONS_ADDR                          0x01A00048 /* Event 3, options */
#define EDMA3_SRC_ADDR_ADDR             0x01A0004C /* Event 3, SRC */
#define EDMA3_ARRYFRAM_ELCNT_ADDR       0x01A00050 /* Event 3, array/frame
count| Event 3, elment count */
#define EDMA3_DST_ADDR_ADDR                                   0x01A00054 /* Event 3, DST
*/
#define EDMA3_ARRYFRAM_ELIDX_ADDR         0x01A00058 /* Event 3, array/frame
index| Event 3, elment index */
#define EDMA3_ELCNTRELD_LINKADDR_ADDR       0x01A0005C /* Event 3, element
count reload| Event 3, link address */

#define EDMA4_OPTIONS_ADDR                          0x01A00060 /* Event 4, options */
#define EDMA4_SRC_ADDR_ADDR             0x01A00064 /* Event 4, SRC */
#define EDMA4_ARRYFRAM_ELCNT_ADDR       0x01A00068 /* Event 4, array/frame
count| Event 4, elment count */
#define EDMA4_DST_ADDR_ADDR                                   0x01A0006C /* Event 4,
DST */
#define EDMA4_ARRYFRAM_ELIDX_ADDR         0x01A00070 /* Event 4, array/frame
index| Event 4, elment index */
#define EDMA4_ELCNTRELD_LINKADDR_ADDR       0x01A00074 /* Event 4, element
count reload| Event 4, link address */

#define EDMA5_OPTIONS_ADDR                          0x01A00078 /* Event 5, options */
#define EDMA5_SRC_ADDR_ADDR             0x01A0007C /* Event 5, SRC */
#define EDMA5_ARRYFRAM_ELCNT_ADDR       0x01A00080 /* Event 5, array/frame
count| Event 5, elment count */
#define EDMA5_DST_ADDR_ADDR                                   0x01A00084 /* Event 5, DST
*/
#define EDMA5_ARRYFRAM_ELIDX_ADDR         0x01A00088 /* Event 5, array/frame
index| Event 5, elment index */
#define EDMA5_ELCNTRELD_LINKADDR_ADDR       0x01A0008C /* Event 5, element
count reload| Event 5, link address */
```

```
#define EDMA6_OPTIONS_ADDR                    0x01A00090 /* Event 6, options */
#define EDMA6_SRC_ADDR_ADDR          0x01A00094 /* Event 6, SRC */
#define EDMA6_ARRYFRAM_ELCNT_ADDR      0x01A00098 /* Event 6, array/frame
count| Event 6, elment count */
#define EDMA6_DST_ADDR_ADDR                        0x01A0009C /* Event 6,
DST */
#define EDMA6_ARRYFRAM_ELIDX_ADDR        0x01A000A0 /* Event 6, array/frame
index| Event 6, elment index */
#define EDMA6_ELCNTRELD_LINKADDR_ADDR       0x01A000A4 /* Event 6, element
count reload| Event 6, link address */

#define EDMA7_OPTIONS_ADDR                    0x01A000A8 /* Event 7, options */
#define EDMA7_SRC_ADDR_ADDR          0x01A000AC /* Event 7, SRC */
#define EDMA7_ARRYFRAM_ELCNT_ADDR      0x01A000B0 /* Event 7, array/frame
count| Event 7, elment count */
#define EDMA7_DST_ADDR_ADDR                        0x01A000B4 /* Event 7,
DST */
#define EDMA7_ARRYFRAM_ELIDX_ADDR        0x01A000B8 /* Event 7, array/frame
index| Event 7, elment index */
#define EDMA7_ELCNTRELD_LINKADDR_ADDR       0x01A000BC /* Event 7, element
count reload| Event 7, link address */

#define EDMA8_OPTIONS_ADDR                    0x01A000C0 /* Event 8, options */
#define EDMA8_SRC_ADDR_ADDR          0x01A000C4 /* Event 8, SRC */
#define EDMA8_ARRYFRAM_ELCNT_ADDR      0x01A000C8 /* Event 8, array/frame
count| Event 8, elment count */
#define EDMA8_DST_ADDR_ADDR                        0x01A000CC /* Event 8,
DST */
#define EDMA8_ARRYFRAM_ELIDX_ADDR        0x01A000D0 /* Event 8, array/frame
index| Event 8, elment index */
#define EDMA8_ELCNTRELD_LINKADDR_ADDR       0x01A000D4 /* Event 8, element
count reload| Event 8, link address */

#define EDMA9_OPTIONS_ADDR                    0x01A000D8 /* Event 9, options */
#define EDMA9_SRC_ADDR_ADDR          0x01A000DC /* Event 9, SRC */
#define EDMA9_ARRYFRAM_ELCNT_ADDR      0x01A000E0 /* Event 9, array/frame
count| Event 9, elment count */
#define EDMA9_DST_ADDR_ADDR                        0x01A000E4 /* Event 9,
DST */
#define EDMA9_ARRYFRAM_ELIDX_ADDR        0x01A000E8 /* Event 9, array/frame
index| Event 9, elment index */
#define EDMA9_ELCNTRELD_LINKADDR_ADDR       0x01A000EC /* Event 9, element
count reload| Event 9, link address */

#define EDMA10_OPTIONS_ADDR                   0x01A000F0 /* Event 10, options */
```

```
#define EDMA10_SRC_ADDR_ADDR            0x01A000F4 /* Event 10, SRC */
#define EDMA10_ARRYFRAM_ELCNT_ADDR      0x01A000F8 /* Event 10, array/frame
count| Event 10, elment count */
#define EDMA10_DST_ADDR_ADDR                      0x01A000FC /* Event 10, DST */
#define EDMA10_ARRYFRAM_ELIDX_ADDR       0x01A00100 /* Event 10, array/frame
index| Event 10, elment index */
#define EDMA10_ELCNTRELD_LINKADDR_ADDR     0x01A00104 /* Event 10, element
count reload| Event 10, link address */

#define EDMA11_OPTIONS_ADDR                       0x01A00108 /* Event 11, options */
#define EDMA11_SRC_ADDR_ADDR            0x01A0010C /* Event 11, SRC */
#define EDMA11_ARRYFRAM_ELCNT_ADDR      0x01A00110 /* Event 11, array/frame
count| Event 11, elment count */
#define EDMA11_DST_ADDR_ADDR                      0x01A00114 /* Event 11, DST */
#define EDMA11_ARRYFRAM_ELIDX_ADDR       0x01A00118 /* Event 11, array/frame
index| Event 11, elment index */
#define EDMA11_ELCNTRELD_LINKADDR_ADDR     0x01A0011C /* Event 11, element
count reload| Event 11, link address */

#define EDMA12_OPTIONS_ADDR                       0x01A00120 /* Event 12, options */
#define EDMA12_SRC_ADDR_ADDR            0x01A00124 /* Event 12, SRC */
#define EDMA12_ARRYFRAM_ELCNT_ADDR      0x01A00128 /* Event 12, array/frame
count| Event 12, elment count */
#define EDMA12_DST_ADDR_ADDR                      0x01A0012C /* Event 12, DST */
#define EDMA12_ARRYFRAM_ELIDX_ADDR       0x01A00130 /* Event 12, array/frame
index| Event 12, elment index */
#define EDMA12_ELCNTRELD_LINKADDR_ADDR     0x01A00134 /* Event 12, element
count reload| Event 12, link address */

#define EDMA13_OPTIONS_ADDR                       0x01A00138 /* Event 13, options */
#define EDMA13_SRC_ADDR_ADDR            0x01A0013C /* Event 13, SRC */
#define EDMA13_ARRYFRAM_ELCNT_ADDR      0x01A00140 /* Event 13, array/frame
count| Event 13, elment count */
#define EDMA13_DST_ADDR_ADDR                      0x01A00144 /* Event 13, DST */
#define EDMA13_ARRYFRAM_ELIDX_ADDR       0x01A00148 /* Event 13, array/frame
index| Event 13, elment index */
#define EDMA13_ELCNTRELD_LINKADDR_ADDR     0x01A0014C /* Event 13, element
count reload| Event 13, link address */

#define EDMA14_OPTIONS_ADDR                       0x01A00150 /* Event 14, options */
#define EDMA14_SRC_ADDR_ADDR            0x01A00154 /* Event 14, SRC */
#define EDMA14_ARRYFRAM_ELCNT_ADDR      0x01A00158 /* Event 14, array/frame
count| Event 14, elment count */
#define EDMA14_DST_ADDR_ADDR                      0x01A0015C /* Event 14, DST */
#define EDMA14_ARRYFRAM_ELIDX_ADDR       0x01A00160 /* Event 14, array/frame
index| Event 14, elment index */
```

```
#define EDMA14_ELCNTRELD_LINKADDR_ADDR     0x01A00164 /* Event 14, element
count reload| Event 14, link address */

#define EDMA15_OPTIONS_ADDR                           0x01A00168 /* Event 15, options */
#define EDMA15_SRC_ADDR_ADDR            0x01A0016C /* Event 15, SRC */
#define EDMA15_ARRYFRAM_ELCNT_ADDR      0x01A00170 /* Event 15, array/frame
count| Event 15, elment count */
#define EDMA15_DST_ADDR_ADDR                          0x01A00174 /* Event 15, DST */
#define EDMA15_ARRYFRAM_ELIDX_ADDR       0x01A00178 /* Event 15, array/frame
index| Event 15, elment index */
#define EDMA15_ELCNTRELD_LINKADDR_ADDR     0x01A0017C /* Event 15, element
count reload| Event 15, link address */

#define EDMA_NULL_PTR1          0x01A007E0 /* Event 15, options */
#define EDMA_NULL_PTR2      0x01A007E4 /* Event 15, SRC */
#define EDMA_NULL_PTR3      0x01A007E8 /* Event 15, array/frame count| Event 15, elment
count */
#define EDMA_NULL_PTR4                    0x01A007EC /* Event 15, DST */
#define EDMA_NULL_PTR5             0x01A007F0 /* Event 15, array/frame index| Event 15,
elment index */
#define EDMA_NULL_PTR6                    0x01A007F4 /* Event 15, element count reload|
Event 15, link address */

/*********************************************************/
//EMIF Control Register
#define GBLCTL_ADDR                              0x01800000    // Global Control Reg
#define CE1CTL_ADDR                              0x01800004    // EMIF CE1 Space Control
Reg
#define CE0CTL_ADDR                              0x01800008    // EMIF CE0 Space Control
Reg
#define CE2CTL_ADDR                              0x01800010    // EMIF CE2 Space Control
Reg
#define CE3CTL_ADDR                              0x01800014    // EMIF CE3 Space Control
Reg
#define SDCTL_ADDR                               0x01800018    // EMIF SDRAM Control
Reg
#define SDTIM_ADDR                               0x0180001C    // EMIF SDRAM Refresh
Control Reg


/*********************************************************/
// Interupt Control Register */
#define MUXH_ADDR                                0x019C0000   // Interupt Multyplexer High
#define MUXL_ADDR                                0x019C0004   // Interupt Multyplexer Low
#define EXTPOL_ADDR                              0x019C0008   // External Interupt Polarity
```

```
/**********************************************************/
// Multi-Channel Buffered Serial Port Conrol Register
// Channel 0
// RBR                  Accessed via CPU/EDMA                   Recieve Buffer Reg
// RSR                  Accessed via CPU/EDMA                   Resiece shift Rg
// XSR                  Accessed via CPU/EDMA                   Transmit Shift Reg
#define CH_0_DRR_ADDR                  0x018C0000   // Channel 0 Data Recieve Reg
#define CH_0_DXR_ADDR                  0x018C0004   // Channel 0 Data Transmit Reg
#define CH_0_SPCR_ADDR                 0x018C0008   // Channel 0 Serial Port Control Reg
#define CH_0_RCR_ADDR                  0x018C000C   // Channel 0 Recieve Control Reg
#define CH_0_XCR_ADDR                  0x018C0010   // Channel 0 Transmit Control Reg
#define CH_0_SRGR_ADDR                         0x018C0014   // Channel 0 Sample Rate
Generator Reg
#define CH_0_MCR_ADDR                  0x018C0018   // Channel 0 Miltichannel Control
Reg
#define CH_0_RCER_ADDR                         0x018C001C   // Channel 0 Recieve Channel
Enable Reg
#define CH_0_XCER_ADDR                         0x018C0020   // Channel 0 Transmit
Channel Enable Reg
#define CH_0_PCR_ADDR                  0x018C0024   // Channel 0 Pin Control

// Channel 1
// Accessed via CPU/EDMA               RBR    Recieve Buffer Reg
// Accessed via CPU/EDMA               RSR    Resiece shift Rg
// Accessed via CPU/EDMA               XSR    Transmit Shift Reg
#define CH_1_DRR_ADDR                  0x019C0000   // Channel 1 Data Recieve Reg
#define CH_1_DXR_ADDR                  0x019C0004   // Channel 1 Data Transmit Reg
#define CH_1_SPCR_ADDR                 0x019C0008   // Channel 1 Serial Port Control Reg
#define CH_1_RCR_ADDR                  0x019C000C   // Channel 1 Recieve Control Reg
#define CH_1_XCR_ADDR                  0x019C0010   // Channel 1 Transmit Control Reg
#define CH_1_SRGR_ADDR                         0x019C0014   // Channel 1 Sample Rate
Generator Reg
#define CH_1_MCR_ADDR                  0x019C0018   // Channel 1 Miltichannel Control
Reg
#define CH_1_RCER_ADDR                         0x019C001C   // Channel 1 Recieve Channel
Enable Reg
#define CH_1_XCER_ADDR                         0x019C0020   // Channel 1 Transmit
Channel Enable Reg
#define CH_1_PCR_ADDR                  0x019C0024   // Channel 1 Pin Control

/**********************************************************/
// Timer Contol Register
// Timer0
#define TIMER0_CTL_ADDR                0x01940000   // Timer0 Control Reg
#define TIMER0_PRD_ADDR                0x01940004   // Timer0 Period Reg
#define TIMER0_CNT_ADDR                0x01940008   // Timer0 Counter Reg
```

```
// Timer1
#define TIMER1_CTL_ADDR              0x01980000    // Timer1 Control Reg
#define TIMER1_PRD_ADDR              0x01980004    // Timer1 Period Reg
#define TIMER1_CNT_ADDR              0x01980008    // Timer1 Counter Reg

#endif
```