UNIVERSITY OF CALGARY

Implantable, Transcutaneously Powered Neurostimulator System To Restore Gastrointestinal Motility

by

Denis Onen

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

FACULTY OF ENGINEERING

CALGARY, ALBERTA

January, 2005

© Denis Onen 2005

THE UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled, "Implantable, Transcutaneously Powered Neurostimulator System to Restore Gastrointestinal Motility" submitted by Denis Onen in partial fulfillment of the requirements for the degree of Master of Science.

M. P. Mintchev

Supervisor, Dr. M. Mintchev,
Department of Electrical and Computer Engineering.

Dr. S.N. Yanushkevich,
Department of Electrical and Computer Engineering.

Dr. E. Sharlin,
Department of Computer Science.

Jan 18, 2005

Date

Abstract

This neurostimulator design research addresses a deficiency of a promising, new stimulation method for imposing a pre-designed and electrically imposed motility pattern in the gastrointestinal tract. The work presented in this thesis was aimed at designing a transcutaneously-powered implantable device to facilitate the testing of this stimulation method.

The outcome of this project is the design and implementation of an innovative transcutaneously-powered device that is small and safe enough to be implanted in an animal. The device conforms to the stimulation algorithm used in the SNGES method to restore gastrointestinal motility.

A detailed design procedure addresses key issues such as charge balance, tissue heating, and biocompatibility. This design procedure lays the foundation for the future architecture of a mixed-signal ASIC based neurostimulator that would lead to further minimization of power consumption and size.

Acknowledgements

I would like to express my sincerest gratitude and appreciation to my supervisor, Dr. Martin P. Mintchev for his guidance and support during the course of my M. Sc. studies. My academic experience was enriched by his professionalism, critical analysis, and dedication to research. I am also grateful to Dr. Mintchev for proposing this research project and accepting me as a student in his laboratory.

I also thank my colleagues in the Low Frequency Instrumentation Laboratory for many interesting technical discussions and also for their friendship. Special thanks go to Ehsan Jalilian for his valuable technical feedback during the course of my research. I wish to express my sincere gratitude to my collaborator, James Doherty, whose ready cooperation and outstanding technical ability greatly facilitated the completion of my research. I extend special thanks to John Shelley for his flawless manufacture of my printed circuit boards.

My thanks also go to my industrial colleagues, whose valuable feedback improved the design decisions that I made in my research. I would especially like to thank my brother, Omer Onen, and my friend, Vladimir Suvorov, for their many valuable discussions.

My sincerest thanks go to my family for their support and love during the course of my research. I thank my wife, Zeynep, for always encouraging me, and my children, Arda and Elif, for their patience while I completed my research. I also would like to thank my mother, Harika, for her love and encouragement.

Dedication

For my wife, with all my love.

Table of Contents

Approval Page ii

Abstract iii

Acknowledgements iv

Dedication v

Table of Contents vi

List of Tables ix

List of Figures and Illustrations x

INTRODUCTION 1

I.1 The Problem of Impaired Gastrointestinal Motility. 1

I.2 Brief Introduction to the Anatomy and Physiology of the Gastrointestinal Wall. 2

I.3 Brief Introduction to the Anatomy and Physiology of the Stomach. 5

I.4 Brief Introduction to the Anatomy and Physiology of the Colon. 6

I.5 Motivation for this Research. 8

I.6 Collaboration. 8

I.7 Contributions of this Thesis 9

I.8 Structure of the Thesis. 9

CHAPTER ONE: METHODS FOR GASTROINTESTINAL ELECTRICAL STIMULATION 11

1.1 Gastric Electrical Stimulation 11

1.1.1 Gastric Electrical Pacing 11

1.1.2 High Frequency Gastric Electrical Stimulation 13

1.1.3 Sequential Neural Gastrointestinal Electrical Stimulation (NGES) 14

1.2 Colonic Electrical Stimulation 17

1.3 Functional Requirements for a Multi-Channel Sequential Stimulation Gastrointestinal Neurostimulator 21

CHAPTER TWO: SYSTEM ARCHITECTURE 23

2.1 Neurostimulator System Architecture 23

2.2 Design Constraints 26

2.3 Design Tradeoff Analysis 28

2.3.1 Power Sources for Implantable Gastrointestinal Neurostimulators 28

2.3.2 Voltage Regulation Methods 29
2.3.3 System Controller 36
2.3.4 Data Transmission Methods 42
2.4 Overview of the Xilinx CoolRunner-II XC2C512 CPLD 46
2.5 Overview of Verilog Programming 51
CHAPTER THREE: DETAILED IMPLANT DESIGN 52
3.1 Detailed Block Diagram of the Neurostimulator System 52
3.2 Clock Source 54
3.3 CPLD Implementation 55
3.3.1 CPLD Design Architecture 55
3.4 Analog Output Voltage Controller 71
3.5 Serial EEPROM 74
3.6 Analog Switch 75
CHAPTER FOUR: TESTING RESULTS 76
4.1 Digital PCB Power Consumption 77
4.2 Implant Power Consumption and Efficiency When Stimulating 78
4.2.1 Colonic Testing Results 79
4.2.2 Worst-Case Gastric Test Results 91
4.2.3 Discussion of Test Results 95
CHAPTER FIVE: CONCLUSIONS 97
BIBLIOGRAPHY 100
APPENDIX A: VERILOG CODE 111
A.1. Module paramFile.v 112
A.2. Module top.v 113
A.3. Module crc.v 117
A.4. Module freqGenerator.v 119
A.5. Module i2c.c 122
A.6. Module manUart.v 130
A.7. Module chSwitcher.v 139
A.8. Module register.v 146
A.9. Module resetBlock.v 148

A.10. Module sysControl.v 149
A.11. Module timer.v 161
A.12. Testbench top_tb.v 162
A.13. Xilinx User Constraint File top.ucf 171
APPENDIX B: SCHEMATIC DIAGRAMS 172
APPENDIX C: PRINTED CIRCUIT BOARD DIAGRAMS 175
APPENDIX D: TABLE OF PROGRAMMBLE SETTINGS TO CONTROL NEUROSTIMULATOR FREQUENCY AND DUTY CYCLE 180
APPENDIX E: MATLAB CODE TO GENERATE OPTIMUM AND FREQUENCY AND DUTY CYCLE PARAMETERS 191
APPENDIX F: TABLE OF PROGRAMMABLE SETTINGS TO CONTROL NEUROSTIMULATOR OUTPUT VOLTAGE 193

List of Tables

Table 1. Stimulation Requirements for a Multi-Channel, Sequential Stimulation Gastrointestinal Neurostimulator 22
Table 2. CPLD Resource Usage for Complete Design 57
Table 3. CPLD Resource Usage for Individual Modules 58
Table 4. Implant Messages to Belt 60
Table 5. Register Map of CPLD 64
Table 6. High and Low Count Values for 50 Hz and Varying Frequencies 70
Table 7. Digital Potentiometer Settings to Control Stimulation Voltage 73
Table 8. Digital Circuit Power Consumption When Idle 78
Table 9. Digital Circuit Power Consumption When Transmitting 78
Table 10 Colonic Test 1, No Stimulation 80
Table 11 Colonic Test 2, No Stimulation 81
Table 12 Colonic Test 3, No Stimulation 82
Table 13 Colonic Test 4, No Stimulation 83
Table 14 Colonic Test 1, Channel 1 On 84
Table 15 Colonic Test 2, Channel 1 On 85
Table 16 Colonic Test 3, Channel 1 On 86
Table 17 Colonic Test 4, Channel 1 On 87
Table 18 Colonic Test 1, Channel 1 and 2 On 88
Table 19 Colonic Test 2, Channel 1 and 2 On 89
Table 20 Colonic Test 3, Channel 1 and 2 On 90
Table 21 Colonic Test 4, Channel 1 and 2 On 91
Table 22 Gastric Test 1, No Stimulation 92
Table 23 Gastric Test 2, No Stimulation 93
Table 24 Gastric Test 1, Channel 1 On 94
Table 25 Gastric Test 2, Channel 1 On 95

List of Figures and Illustrations

Figure 1. Cross-Section of the Gastrointestinal Wall (taken with permission from Mintchev, 2005) 3
Figure 2. The Gastrointestinal Tract 5
Figure 3. Electrode Placement for Gastric SNGES 16
Figure 4. Sample Stimulation Pattern for SNGES 16
Figure 5. Electrode Placement for Colonic SNGES 20
Figure 6. High-level Block Diagram of Neurostimulator System (Colonic Example) 24
Figure 7. Linear Voltage Regulator 31
Figure 8. Step-Down Switching Regulator: Energizing the Inductor 33
Figure 9. Step-Down Switching Regulator: De-energizing the Inductor 34
Figure 10. Minimum Timing Interval Between Events 39
Figure 11. UART Protocol 45
Figure 12. Example of Manchester Encoded Data 46
Figure 13. CoolRunner-II CPLD Architecture (Xilinx, 2004a) 47
Figure 14. CoolRunner-II CPLD Function Block (Xilinx, 2004a) 48
Figure 15. Basic PLA Architecture (Xilinx, 2002) 49
Figure 16. CoolRunner-II CPLD Macrocell (Xilinx, 2004a) 50
Figure 17. Detailed Block Diagram of Neurostimulator System (Colonic Example) 52
Figure 18. Implant CPLD Design Architecture 56
Figure 19. Block Diagram of sysControl.v 58
Figure 20. Block Diagram of manUart.v 61
Figure 21. Block Diagram of register.v 63
Figure 22. Block Diagram of crc.v 65
Figure 23. Block Diagram of i2c.v 65
Figure 24. Digital Potentiometer I^2C Read/Write Protocol (Analog Devices, 2004a) 66
Figure 25. Serial EEPROM I^2C Read/Write Protocol (Microchip, 2003) 67
Figure 26. Block Diagram for freqGenerator.v 68
Figure 27. Stimulation Signal Synthesis with the High/Low Count Parameters 69
Figure 28. Block Diagram for chSwitcher.v 71
Figure 29. Circuit Diagram of Adjustable Voltage Regulator Programming 72

Figure 30. Test Set-Up to Measure Digital Circuit Power Consumption 77
Figure 31. Test Set-Up to Measure Stimulation Power Efficiency 79

INTRODUCTION

I.1 The Problem of Impaired Gastrointestinal Motility.

In the past few years there has been a growing interest in treating a variety of functional diseases of gastric and colonic transport (O'Brien & Phillips, 1996) (Folwaczny et al., 1999) using neural electrical stimulation of gastrointestinal organs (Hollerbach et al., 1997), especially of the stomach and colon. Millions of Canadian men, women, and children of all ages are affected by functional gastrointestinal (GI) disorders. These disorders are the most commonly encountered gastrointestinal illnesses seen by physicians in primary care or gastroenterology. Irritable bowel syndrome (IBS) and severe chronic constipation are among the most common of these disorders, with IBS affecting 10% to 20% of adults. Furthermore, about 0.1% of the population is affected by drug-refractory diabetic or idiopathic gastroparesis, a severe functional disorder of the stomach related whereby it cannot propel content to the gut. This disorder seriously impacts the patients' lifestyle and well-being, plus it burdens the medical system with a significant cost. The social and economic costs of functional gastrointestinal disorders are enormous. Patients exhibiting severe symptoms can be debilitated by these disorders, unable to fully partake in life and work, especially where surgery may result in the removal of a significant portion of the gut. These disorders are still not well understood, causing confusion and misunderstanding among patients and physicians, and sometimes resulting in misdiagnosis and inappropriate treatment. For example, there is a lack of non-invasive therapy for IBS or chronic idiopathic constipation. Data has revealed that women with these disorders, therefore, risk unnecessary or unneeded surgery. Disorders such as colonic pseudo-obstruction, IBS, GI dysmotility in severe cases of diabetes

(Folwaczny et al., 1999), and particularly *chronic idiopathic constipation and drug-refractory gastroparesis* can be not only devastating for the patient lifestyle but lead to severe life-threatening complications. Despite the negative impact that gastroparesis and chronic constipation have on the quality of life for affected individuals, these disorders are not correlated with a significant mortality rate, with the 5 year mortality rate for patients with gastroparesis being 5% (McCallum & George, 2001). For diabetic gastroparesis, 9 years after the first positive test for delayed gastric emptying, the mortality rate is 29% for patients with delayed solid emptying and 25% for patients with delayed liquid emptying (Kong et al., 1999). Furthermore, it has been found that there are no marked changes in emptying for diabetic gastroparesis over a 12-year period (Jones et al., 2002). Although 4.4 million individuals in the United States suffer from constipation, it results in only 29 deaths per year (Digestive Disease Statistics, 1999). External electrical stimulation of impaired propulsive or segmental gastrointestinal motility may provide a successful long-term treatment of such disorders (Huizinga, 1986). It can also be extremely useful in shortening postoperative ileus (Livingston & Passaro, 1990), thus helping to significantly decrease postoperative hospitalization and reduce health-related costs.

I.2 Brief Introduction to the Anatomy and Physiology of the Gastrointestinal Wall.

The walls of the stomach and the colon are similar to most other parts of the gastrointestinal tract. Therefore, only a general view of the gastrointestinal tract is shown in Figure 1 (Marieb, 1998). The function of the gastrointestinal wall is to contain, mechanically process, and propagate gastrointestinal contents. The wall is comprised of 4

layers of tissue: the mucosal layer; the submucosal layer; the muscularis; and the serosal layer. The open space or cavity enclosed by the gastrointestinal wall is referred to as the lumen.

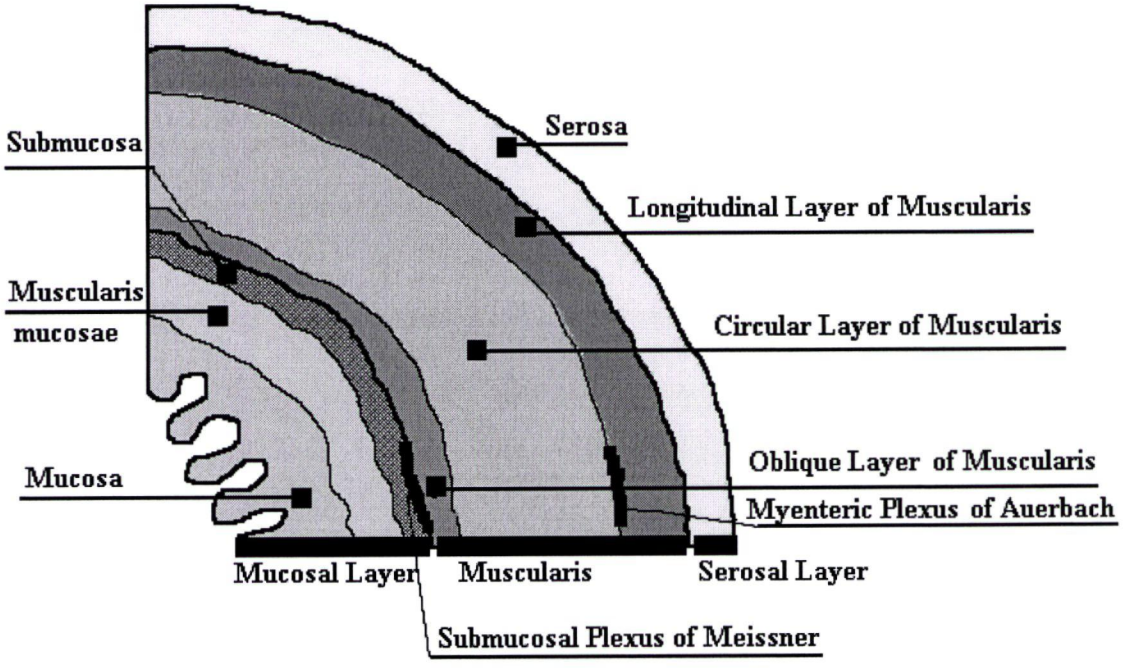


Figure 1. Cross-Section of the Gastrointestinal Wall (taken with permission from Mintchev, 2005)

The mucosal layer faces the lumen and is comprised of the mucosa, an epithelial lining, and the muscularis mucosae, a layer of smooth muscle. Both of these layers are separated by a layer of loose connective tissue. The submucosa is mostly connective tissue but also contains small glands, blood vessels, and nerves of the autonomic nervous system. These nerves are the submucosal plexus of Meissner. The muscularis has

generally two layers of smooth muscle but the stomach has a third layer. These layers are defined by the direction of the muscle fibers in the layer. The outer, longitudinal layer consists of muscle cells arranged parallel to the luminal axis. The inner, circular layer consists of muscle cells arranged circumferentially to the luminal axis. The nerves of the myenteric plexus of Auerbach are between the circular and longitudinal muscle layers. The stomach has a further inner, oblique layer whose muscle cells are arranged diagonally to the luminal axis. This oblique layer exists mostly in the fundus and the corpus (Williams, 1995). The serosal layer covers the gastrointestinal organs.

The digestive tract is responsible for the ingestion, digestion, and absorption of food, and the elimination of indigestible matter such as feces (Johnson, 1994). The process of digestion modifies the chemical and physical composition of food to a form that can be absorbed by the body. Digestive juices mix with the food in the lumen and this mixture is ground and churned by motility patterns in the gastrointestinal tract. Absorption of the luminal contents by the body occurs in the intestines. The colon moves indigestible matter to the rectum with powerful propagating contractile patterns, for expulsion through the anus.

Peristalsis and segmentation are the two main types of gastrointestinal motility patterns (Devinder et al., 1988). Peristalsis is the progressive, wavelike, involuntary contractions generated by the gastrointestinal tract used to move contents to different locations along the tract. Circular contractions move contents by generating increased pressure at the contraction site, which distends the adjacent section of the gastrointestinal wall as contents flow to the adjacent section. Subsequent and repeated circular contractions at the distended sections propagate contents through the gastrointestinal

tract. Segmentation is the back and forth movement of gastrointestinal contents by random contractile activity in the gastrointestinal tract. This motility pattern helps grind and mix food with digestive juices. This pattern also exposes the food to the mucosa for absorption.

The interstitial cells of Cajal are located in most parts of the digestive system, including the stomach and colon (Hanani et al., 2000). They are thought to be the pacemaker of the digestive system, generating the slow wave that controls the motility of the gastrointestinal tract.

I.3 Brief Introduction to the Anatomy and Physiology of the Stomach.

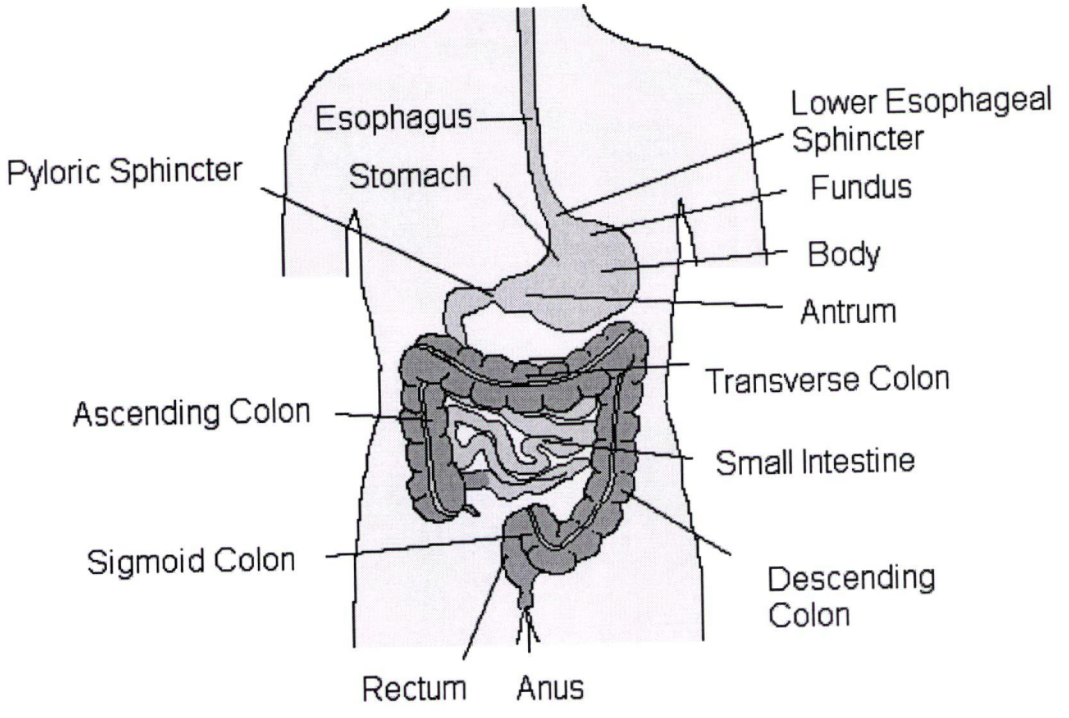


Figure 2. The Gastrointestinal Tract

The stomach is an elongated pouchlike digestive organ located below the diaphragm, between the esophagus and the small intestine. Its size depends on the amount of food within it, and ranges from 1 to 1.5 L in adults (Williams, 1995). The stomach is isolated from the rest of the digestive tract by two sphincters. The Lower Esophageal Sphincter (LES) isolates the stomach from the esophagus and the pyloric sphincter isolates the stomach from the small intestine. Anatomically, the stomach can be divided into three major regions: fundus, body, and antrum. The most proximal part is the fundus and it is to the left and above the esophageal opening. The middle of the stomach is the body. The fundus is histologically very similar to the body (corpus). The antrum is the most distal part of the stomach and it is where the density of smooth muscle increases. The circumferences of the fundus and body are greater than that of the antrum.

The primary function of the stomach is to process and move food to the small intestine (Devinder et al., 1988). Gastric peristaltic contractions churn and grind food together with gastric juice (mainly hydrochloric acid, pepsin, rennin, and mucin) secreted by the stomach to convert it to chyme. This partially digested chyme has a souplike consistency and is passed to the small intestine for the absorption of nutrients. Typically, the stomach empties within approximately 2 to 6 hours after a meal. The stomach uses rhythmic peristaltic contractions at a rate of about 3 cycles per minute (about 20 seconds) to move the contents through the pylorus.

I.4 Brief Introduction to the Anatomy and Physiology of the Colon.

The ileal valve connects the small intestine to the large intestine. The large intestine can be divided into two main parts, the proximal division is the cecum and the distal

division is the colon (Phillips, 1991). The colon is approximately 1.5 m in length for an adult and essentially follows the perimeter of the abdominal cavity. There are four anatomical divisions of the colon, the ascending colon, transverse colon, descending colon, and sigmoid colon. The ascending colon rises vertically on the right side of the abdomen from the cecum towards the liver. The transverse colon crosses the abdomen horizontally from right to left, below the liver, stomach, and spleen. The descending colon travels vertically down the left side of the abdomen to the sigmoid colon in the pelvis. The sigmoid colon connects to the rectum. The ascending colon is joined at a right angle to the transverse colon by the hepatic fixture, which is affixed to the abdominal cavity to support the colon. The splenic fixture joins the transverse colon to the descending colon at a right angle and fixes it to the abdominal cavity as a second point of support.

The main functions of the colon are to: mix intestinal content; absorb water; and to form, pack, store, and evacuate indigestible matter such as feces (Devinder et al., 1988). Segmentation contractions occur mainly in the ascending and transverse sections of the colon. These colonic sections also absorb potassium and sodium ions from the luminal contents during the churning action from segmentation. Water from the luminal contents is absorbed in the transverse and descending colonic sections. Very powerful peristaltic contractions move the solid content from the descending colon to the sigmoid colon and then to the rectum. These powerful peristaltic waves typically occur 3 to 4 times per day. The sigmoid colon stores the packed and formed feces to decrease rectal pressure until elimination. Synchronized contractions of the sigmoid colon and rectum assist in creating

increased intraluminal pressure on the anus to facilitate expulsion of feces, with a corresponding relaxation of the anus.

I.5 Motivation for this Research.

The present research was motivated by the success of our research group in developing an innovative method for microprocessor-based restoration of impaired gastrointestinal motility using sequential high-frequency neural gastrointestinal electrical stimulation (NGES). A thorough description of this technique and its place with respect to other existing methods for gastrointestinal stimulation is offered in Chapter 1 of the present thesis. A major deficiency of this newly proposed and very promising method for imposing a pre-designed and electrically induced motility pattern in the gut (Bortolotti, 2002), was the fact that a transcutaneously-powered implantable device implementing such stimulation protocols was lacking. Thus, my M.Sc. work aimed at filling that important technological void, precluding the testing of the method on chronic experimental animals and subsequently on humans.

I.6 Collaboration.

This research effort to design the transcutaneously powered neurostimulator system was a successful collaborative effort between another graduate student, James Doherty and me, under the leadership and overall direction of Dr. Martin Mintchev. The primary design responsibilities were divided between the neurostimulator and the transcutaneous power and communications link. I was responsible for the neurostimulator design and Mr. Doherty was responsible for the transcutaneous power and

communications link. System integration and testing were performed jointly by Mr. Doherty and me.

I.7 Contributions of this Thesis.

This thesis presents the design of the first published implantable, battery-less, multi-channel, sequential gastrointestinal neurostimulator. The sequential, multi-channel stimulation technique has previously been shown to be an extremely promising method in restoring gastrointestinal motility. This neurostimulator will facilitate the laboratory testing of this method on animals and eventually, humans. The integration of the transcutaneous power transfer design with the neurostimulator represents advancement in the long-term care of patients with implantable electrical gastrointestinal stimulators because this method eliminates the implantable battery, which has a finite life, often less than the duration of the disorder the implant is meant to address.

I.8 Structure of the Thesis.

Chapter 1 describes and critiques various electrical stimulation methods that have been explored in literature for the restoration of impaired gastrointestinal motility, applied to the stomach and colon. From this survey, the basic functional design requirements of the neurostimulator are summarized. Chapter 1 concludes with a discussion on design implications for implantable neurostimulators with batteries or with a transcutaneous power transfer or a combination of both.

Chapter 2 describes the overall system architecture and an analysis of the design tradeoffs involved in various implementation strategies of the neurostimulator. From this

analysis, a detailed set of requirements are developed to reflect the selected implementation approach.

Chapter 3 describes the detailed design of the neurostimulator, including the devices, important calculations, and Verilog code. The limitations of the design are also discussed.

Chapter 4 describes the results of testing the neurostimulator, including: energy consumption for typical usage profiles; and power efficiency.

Chapter 5 concludes the thesis and includes recommendations for future research.

Appendix A contains the Verilog source code for the design. Appendix B contains the schematic diagrams. Appendix C contains the printed circuit board diagrams. Appendix D contains a table of programmable frequencies and duty cycles. Appendix E contains Matlab code to generate the programmable frequencies and duty cycle table. Appendix F contains a table of programmable voltage.

CHAPTER ONE: METHODS FOR GASTROINTESTINAL ELECTRICAL STIMULATION

Since the pioneering work of Bilgutay et al. (1963) on gastrointestinal (GI) stimulation, the feasibility of this stimulation technique has been extensively explored. This chapter presents a brief review of the methods for gastric electrical stimulation and the ones for colonic electrical stimulation. The abundant scientific literature is presented in two separate subsections so that a clear delineation is made between the two subjects. Some of the discussed methods could be applicable to other portions of the gut (e.g., the esophagus, the duodenum, and the small intestine). However, for the purpose of this research, and considering the epidemiology and severity of gastrointestinal motility disorders, the applicability of these techniques is focused on the stomach and the colon.

1.1 Gastric Electrical Stimulation

Any electrical stimulation method applied to the stomach is referred to as gastric electrical stimulation (GES). In the following subsections, specific GES methods are classified and outlined.

1.1.1 Gastric Electrical Pacing

Gastric electrical pacing (GEP) is a method to induce entrainment of the intrinsic slow wave of the stomach with a frequency at or slightly above the intrinsic gastric slow wave frequency. In a healthy stomach, the intrinsic slow wave is spontaneous and GEP is intended to artificially entrain it in patients with dysfunctional stomachs, presuming that

the major underlying factor impairing the spontaneously-existing motility is a combination of gastric electrical dysrhythmia and lack of electrical synchronization of gastric slow wave in different sections of the organ. It is hypothesized that the electrical stimulation in GEP directly stimulates the interstitial cells of Cajal, or the muscle cells, or both. It is known that intramural cholinergic nerves are not stimulated by GEP. In GEP, the gastric wall is given a stimulus of a 30 to 550 ms square wave pulse at constant current of 2 to 6 mA for a repetition frequency of 3 to 4.7 cycles per minute. Some studies have used voltage-based stimulation in the range of 1 to 25 V instead of constant current. Studies have shown that GEP is able to entrain the intrinsic gastric slow wave, however, this method does not guarantee improvement in gastric emptying (Bortolotti, 2002).

GEP is also known in the literature as low-frequency stimulation (LFS) (Lin et al., 2003) and also as a high-energy, low-frequency stimulation method (McCallum & George, 2001). The major drawback of this method is embedded in its very hypothesis, suggesting that impaired motility in necessarily and directly related to gastric electrical dysrhythmias. The problem is, that global gastric electrical dysrhythmias (1) are relatively rare phenomena in humans; and (2) do not necessarily cause profound and sustained gastric dysmotility. An attestation to that is the fact that although a complete entrainment of the gastric slow wave in orad direction can be achieved, this does not lead to acceleration or improvement in the gastric emptying (Bellahsene et al., 1992), (Eagon & Kelly, 1995), (Hocking et al., 1992).

1.1.2 High Frequency Gastric Electrical Stimulation

High-frequency gastric electrical stimulation (HI-GES) is a method to stimulate the gastric wall with an electrical frequency of that is 4 to 40 times higher that the IGF. In HI-GES, the stimulus has a 2 to 5 mA pulse with width 300 to 330 μs and a frequency of 12 to 30 cycles per minute. HI-GES is thought to stimulate the "sensory fibers directed to the central nervous system." (Bortolotti, 2002). Studies have shown that HI-GES has very little or no effect on the intrinsic gastric slow wave and that there are contradictory results regarding gastric emptying. These studies also demonstrate that there is some reduction in nausea and vomiting (Bortolotti, 2002). Lin et al. (2003) indicate that HI-GES has insufficient intensity to entrain the gastric slow wave or cause contractions of the smooth muscle. A recent study by Lin et al. (2004) suggests that there is a significant postprandial increase in the amplitude of the intrinsic gastric slow wave but no improvement in gastric emptying.

HI-GES has been clinically available from Medtronic Inc. since its Food and Drug Administration (FDA) approval as a humanitarian use device (HUD) in 1999 (Enterra Therapy). Medtronic has published a brochure (Enterra Therapy for Gastroparesis) that indicates a drastic improvement for gastric emptying using HI-GES according to a worldwide anti-vomiting electrical stimulation study (WAVESS). In this brochure, it is stated that the reduction in gastric retention 4 hours after a meal for gastroparetic patients was 81% for diabetic patients and 27% for idiopathic patients after a 12-month period using HI-GES. The figures in the brochure are cited from a 2001 internal (unpublished) Medtronic report on the ongoing WAVESS clinical study. The WAVESS study was published by Abell et al. (2003) in the leading peer-reviewed

journal Gastroenterology and in comparison, this data shows 65% and 18% reductions, respectively, in gastric retention 4 hours after a meal for the diabetic and idiopathic gastroparetic patients. During the course of the WAVESS study, 25 patients were receiving antiemetic therapy and 28 patients were prokinetic therapy, therefore, the effect of HI-GES on symptoms and gastric emptying cannot be isolated from the medication in this study. Another problem with the WAVESS study is the attrition of patients during the course of the study. The study was initiated with 33 patients and only 26 were available for the 6-month follow-up and only 20 were available for the 12-month follow-up, thus limiting the meaningfulness of the statistics in the study. Furthermore, it is possible that the attrition of patients in the WAVESS study may have inadvertently skewed the results in a manner that is favourable to the use of HI-GES for the treatment of gastroparesis. Of the patients removed from the study at 6 and 12 months, vomiting prevented 3 from completing the gastric retention test, 2 refused testing, and 1 was too ill to follow-up. It is possible that this subset of removed patients may have presented test results that were less favourable for the use of HI-GES in the treatment of gastroparesis.

HI-GES is also known in the literature as high-frequency stimulation (HFS) (Lin et al., 2003) and also as a low-energy, high-frequency stimulation method (McCallum et al., 2001).

1.1.3 Sequential Neural Gastrointestinal Electrical Stimulation (NGES)

Neural gastric electrical stimulation (NGES) is a method to produce a local contraction in the stomach in the vicinity of a given electrode site using a long duration pulse train with a frequency 20 to 500 Hz. The first application of this very high

frequency electrical stimulation method to the gastrointestinal tract was by Bilgutay et al. (1963), and was suggested as beneficial in post-operative ileus. NGES is thought to induce an acetylcholine release from the intramural cholingeric fibers, which causes a muscular contraction in the vicinity of the implanted electrodes. In NGES, a pulse train of 4 to 16 seconds with frequency 20 to 500 Hz and repetition frequency of pulse trains of 0.2 to 2 per minute at a current strength of 2 to 10 mA is applied to the gastric wall (Mintchev & Bowes, 1997), (Bortolotti, 2002). One study has used pulse trains of 5 to 10 minutes duration (Eagon & Kelly, 1995).

NGES generally had been unsuccessful in producing propagated gastric contractions until sequential neural gastric electrical stimulation (SNGES) was recently introduced (Mintchev & Bowes, 1997), (Mintchev et al., 2000). In SNGES, sequential sets of electrodes are circumferentially placed as rings from the proximal to distal stomach. The stimulation pattern that has been used is a bipolar voltage pulse train of 4 to 16 seconds in length with frequency 50 Hz and a repetition frequency of pulse trains of 2 cycles per minute at a voltage of 4 to 14 V. This stimulation pattern has been applied sequentially to the sets of ring electrodes. It was shown in the SNGES studies that gastric contractions were induced and that both solids and liquids were propelled through the stomach (Mintchev et al., 1998), (Mintchev et al., 2000). In SNGES, any intrinsic gastric electrical activity is overwhelmed and a specific, programmed contractile pattern is impressed upon the organ in a "brute force" approach to restore gastric motility (Hasler, 2000) in sequentially administered and microprocessor-controlled fashion.

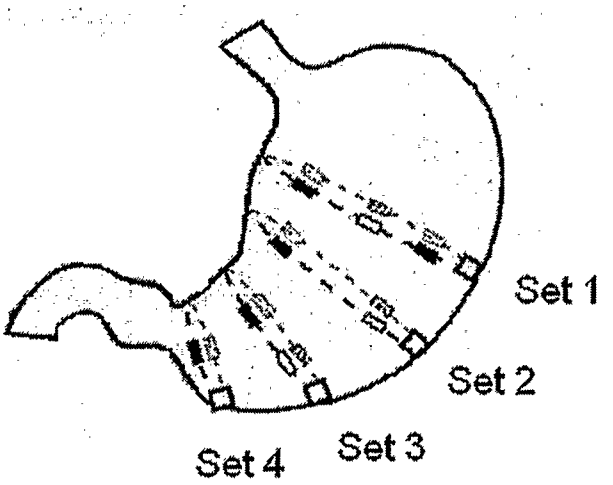


Figure 3. Electrode Placement for Gastric SNGES

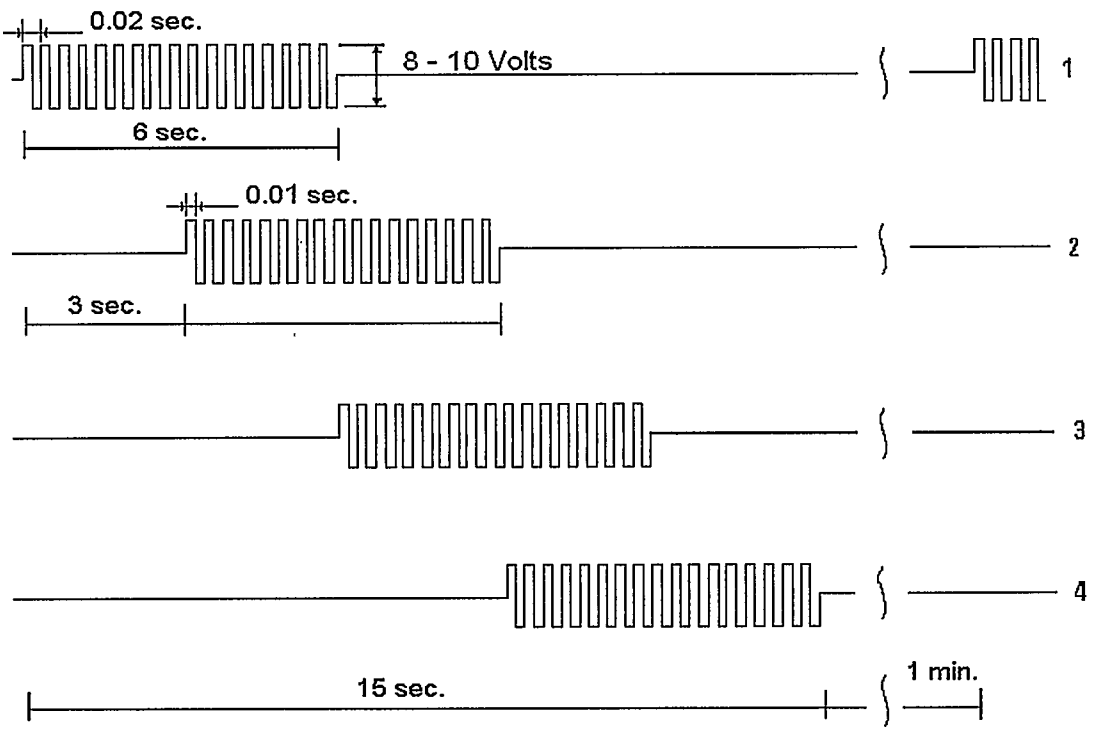


Figure 4. Sample Stimulation Pattern for SNGES

Critical to the successful implementation of SNGES (or any of the other methods, for that matter) is the selection of optimal parameters related to the stimulus pattern required for a particular patient. The modelling of the stomach (Rashev et al., 2002b) and the colon (Rashev et al., 2001) was performed by our research group using generalized cylinders to model the effect of external neural electrical stimulation on gastrointestinal tissue. This modelling takes into account the thickness of the gastrointestinal wall, the electrode dimensions and placement, and the dimensions of the stomach or colon segment being stimulated, and the stimulation pattern itself. The model predicts the occlusion behaviour of the tissue as sequential neural stimulation is applied.

1.2 Colonic Electrical Stimulation

To test the applicability of electrical stimulation in enhancement or mitigation of intestinal propulsion, a number of experiments have been performed using (1) slow wave pacing (Sarr et al., 1981) or (2) controlled emptying of intestinal reservoirs using high frequency (HF, above 10 Hz) direct electrical stimulation of the local nervous system (Grundfest-Broniakowski et al., 1988a), (Hughes et al., 1995).

However, artificial restoration of the mechanical functionality of the human colon using HF stimulation remains an unresolved issue. The considerable experimental work on intestinal functional stimulation has demonstrated the feasibility of controlled emptying of intestinal reservoirs (Grundfest-Broniakowski et al., 1988b) (Hughes et al., 1995) (Grundfest-Broniakowski et al., 1988a), anorectal reconstruction (Ratani et al., 1997), enhancement or mitigation of intestinal propulsion (Sarr et al., 1981) (Cranley et al., 1983) (Varma et al., 1986), and controlled defecation (MacDonagh et al., 1990)

(Bruninga et al., 1998). Three conceptually different approaches for stimulating intestinal motility have been explored in human and animal models.

The first method utilizes the parasympathetic autonomic nervous system pathways to the sigmoid colon, rectum and external anal sphincter by stimulating the anterior sacral roots (S2 – S4) with narrow (0.1 – 0.6 ms) pulses of HF (10 - 100 Hz in humans, Varma et al., 1986) using a Finetech-Brindley stimulator (Brindley et al., 1988) to indirectly enhance intestinal motility. Controlled studies performed in patients with spinal cord lesions indicate significantly improved ability (Cranley et al., 1983) and frequency (Binnie et al., 1991) of defecation, as well as facilitated colonic transit (Binnie et al., 1991). Yet, this type of electrical stimulation involves neurosurgery and deafferentiation of the sacral roots resulting in loss of reflex defecation (Bruninga et al., 1998), which might be unacceptable in cases not involving permanent spinal cord injuries.

The second method for intestinal electrical stimulation employs pacing of the spontaneous electrical control activity of the gut with current pulses of duration 50 ms at infralow frequency (0.25 Hz in canine models) using a pair of electrodes implanted in the serosa to entrain the intestinal electrical activity at the frequency of stimulation (Akwari et al., 1975). It has been shown that pacing can facilitate the movement of liquids within the canine small intestine but it fails to affect significantly the propulsion of solids (Sarr et al., 1981).

Intestinal smooth muscle stimulation utilizing the third method is mediated by the local nervous system, but is accomplished using direct low and high frequency (2 – 40 Hz) voltage stimulation, with pulse duration no longer than half of the stimulation semi-period. Applicability of this approach for emptying intestinal pouches (Grundfest-

Broniakowski et al., 1988b) (Ratani et al., 1997) and lowest portions of the rectosigmoid (Bruninga et al., 1998) has been demonstrated in animal models. However, no attempts have been reported for controlled movement of content within the bowel. In addition, prolonged and repetitive propagation of the produced contractions has not been reported.

A new, fourth method of intestinal stimulation, involving synchronously invoked phasic contractions for the recreation of peristalsis produced under microprocessor control has been recently suggested (Lin et al., 2003), (Rashev et al., 2001), (Amaris et al., 2002). The method originates from the research group to which I presently belong, and in its essence replicates the success of the SNGES technique in producing controlled propagated gastric contractions. In fact, the term SNGES is completely applicable for the colon as well, just broadening its scope from purely gastric to more general gastrointestinal applications. Applying SNGES to sets of implanted colonic electrodes could in fact produce controlled movement of colonic content in acute animal studies (Rashev, Amaris, & Bowes, 2002) (Rashev, Amaris, Bowes, & Mintchev, 2002). Therefore, the synchronization of the sequential stimulating patterns, the shape and the duration of the stimulating voltage, and the positions of the stimulating electrode sets are pivotal for the externally-controlled recreation of impaired gastrointestinal motility.

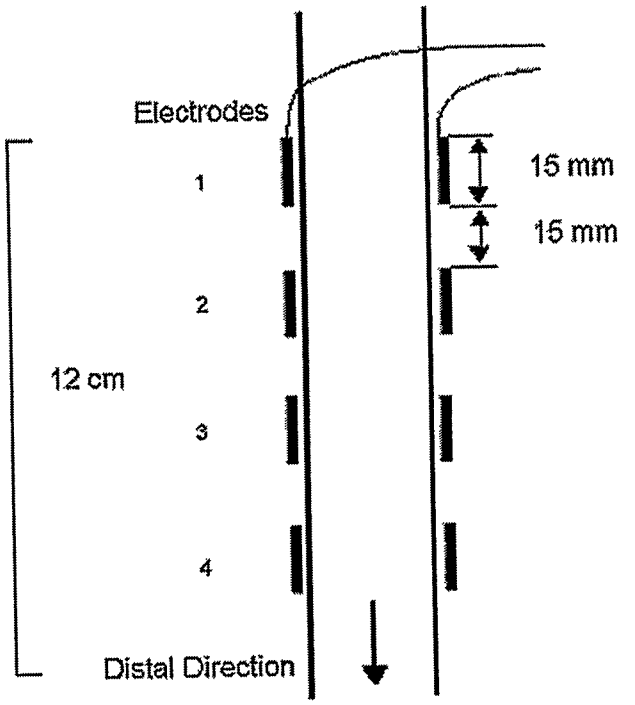


Figure 5. Electrode Placement for Colonic SNGES

The protocol for this stimulation methodology employs HF voltage stimulation (50 Hz) with 10-ms pulses (equal to the half of the stimulation semiperiod) applied to sets of subserosal stimulating electrodes (SEs) implanted circumferentially along the colonic axis, and activated sequentially starting from the proximal to the distal electrode set (forward propulsion) or vice-versa (backward propulsion).

Of the methods for colonic and gastric electrical stimulation, no reports cited the impedance of the tissue being stimulated, except for a study on colonic SNGES (Sanmiguel et al., 2004), to be published in *Gut*, a leading gastroenterological journal. The absence of the impedance measurement makes it impossible to calculate the energy requirements of the stimulation method. Sanmiguel et al. delivered rectangular bipolar

voltage pulses of amplitudes 8 to 10 V peak-to-peak at 50 Hz with 100% duty cycle for 6 seconds to four pairs of electrodes implanted subserosally on the longitudinal axis of a canine colon. The 6-second pulse trains were sequential and overlapped by 3 seconds. This stimulation pattern was repeated after 1 minute and this stimulation protocol was delivered twice daily (i.e. the simulation pattern was delivered 4 times per day). The reported average tissue impedance was 324 Ω ± 26 Ω and the effective current was 7.8 to 9.9 mA. The energy requirements from Sanmiguel et al. can be computed as follows. The average current while stimulating is, $\frac{7.8mA+9.9mA}{2}=8.85mA$. The average power is, $(8.85mA)^2 \cdot 324\Omega = 25.4mW$. The average energy per session (4 channels with duration 6 s each) is, $4 \cdot 6s \cdot 25.4mW = 0.61J$. The daily energy requirement (4 sessions per day) is, $4 \cdot 0.61J = 2.44J$.

1.3 Functional Requirements for a Multi-Channel Sequential Stimulation Gastrointestinal Neurostimulator

From the previously cited literature, the functional requirements for the gastrointestinal neurostimulator are summarized in Table 1.

Table 1. Stimulation Requirements for a Multi-Channel, Sequential Stimulation Gastrointestinal Neurostimulator

Parameter	Unit	Range	Optimal Values (for Colonic stimulation)	Step Size/Resolution
Amplitude	V	1 –10	6	1
Frequency	Hz	30 – 250	50	10
Duty Cycle	%	10 – 100	100	10
Pulse Train Duration	s	1 – 24	6	3
Phase Lag Between Channels	s	1 – 54	3	3
Pause After Stimulation	s	1 – 180	60	20
Overall Pattern (Embedded or Sequential)		E or S		

The "Embedded" stimulation pattern indicated in the bottom of Table 1 is a pattern whereby the 4 channels are activated with a sequential pattern, however, once a channel is activated, it remains active until all the channels are simultaneously deactivated.

CHAPTER TWO: SYSTEM ARCHITECTURE

This chapter introduces the high-level description of the neurostimulator system and the basic design tradeoffs considered in the design and implementation of the neurostimulator. This chapter concludes with a detailed description of the design requirements for the neurostimulator and a description of some of the technologies used.

2.1 Neurostimulator System Architecture

A basic block diagram of the neurostimulator system outlining the main functional blocks, is shown in Figure 6.

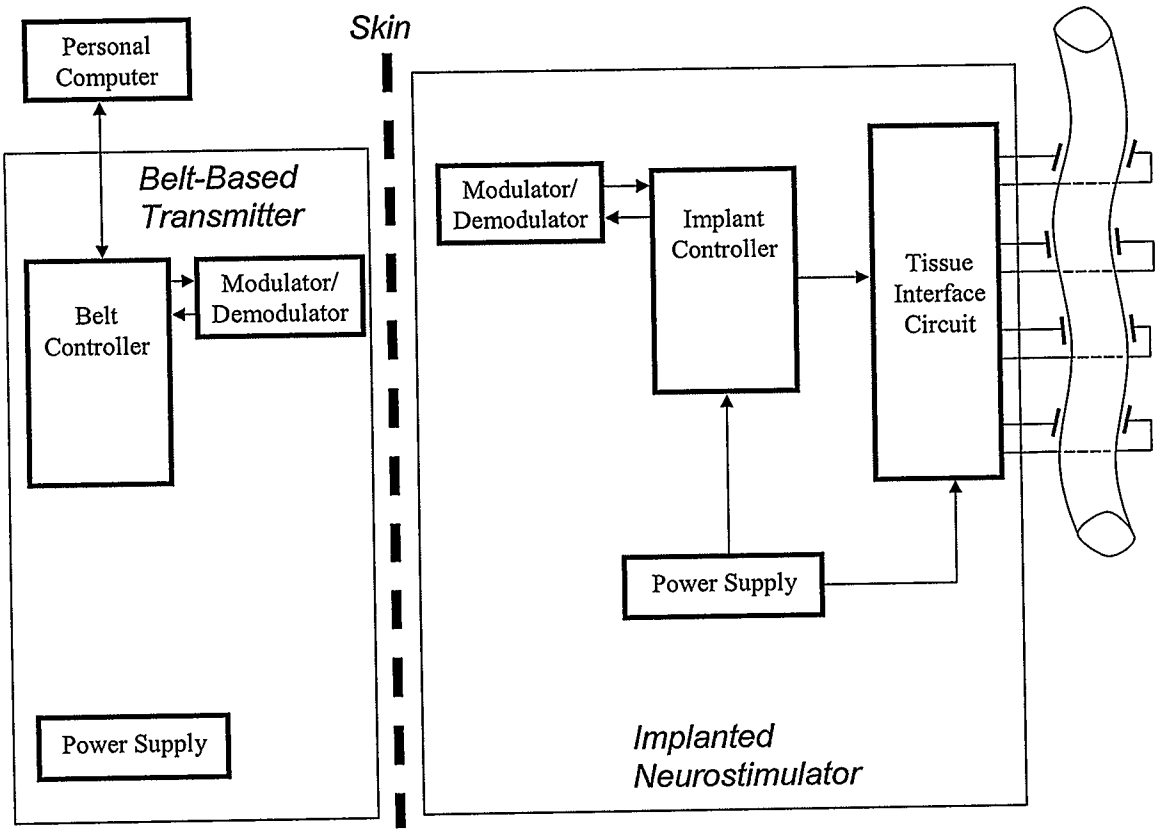


Figure 6. High-level Block Diagram of Neurostimulator System (Colonic Example)

The neurostimulator system has 2 main components: the belt-based transmitter; and the implanted neurostimulator. In this thesis, the belt-based transmitter is referred to as the belt and the implanted neurostimulator is referred to as the implant. The belt is be worn as an ordinary belt on the outside of the body, the implant is implanted subcutaneously between the skin and the abdominal wall. The stimulating electrodes from the implant pass through the abdominal wall and the peritoneum and are inserted subserosally to the gastrointestinal organ.

The belt contains:

1. a modulator and demodulator which wirelessly send and receive data to and from the implant;
2. a power supply which provides energy to the belt circuitry;
3. a controller which:
    a. transmits configuration data to the implant using the modulator;
    b. receives status data from the implant using the demodulator;
    c. communicates with a personal computer, receiving configuration data to be sent to the implant and sending status information to the computer from the implant.

The implant contains:

1. a power supply which provides energy to the implant;
2. a modulator and demodulator which send and receive data to and from the belt;
3. a tissue interface circuit which drives the electrical stimulation signals to the gastrointestinal organ;
4. a controller which:
    a. receives configuration data from the belt using the demodulator;
    b. sends status data to the belt using the modulator;
    c. generates the control signals to the tissue interface circuit.

2.2 Design Constraints

Functionally, the implant must meet the electrical and timing requirements of the method, as outlined in Table 1. Several safety factors also constrain the design. The safety factors include: temperature elevation of the tissue adjacent to the implant; electrical charge balance of the electrical stimulation to the tissue; and the biocompatibility of the implant enclosure.

Heating of the tissue adjacent to the implant is an important problem and a difficult one to model. Tissue heating is related to many factors, which include: the energy dissipated as heat due to circuit inefficiencies; the thermal coupling of the implant to the tissue; and the heat removal due to blood flow, amongst other factors (Uzunoglu & Nikita, 1988). Generally, for implants, thermal elevation due to heating is considered safe if it is 2 degrees Celsius or less (Wise et al., 2004), (Hrdlicka, 2002). The problem of tissue heating was addressed in the research presented in this thesis by following low power design strategies for the circuit design. However, tissue heating was not considered to be a critical design problem because of the short duration (e.g. several minutes at most) of the stimulation period. In acute and chronic animal testing, the effect of tissue heating will be monitored by visual inspection of the tissue adjacent to the implant at the conclusion of the experiment, followed by histological analysis, if warranted.

Accumulation of charge needs to be avoided when electrically driving electrodes in tissue (Plonsey & Barr, 1988), (Wise et al., 2004). An excessive charge can result in tissue damage when a negative voltage is applied to the tissue for an extended period of time. Similarly, an excessive charge can result in electrode damage when a positive voltage is applied to the tissue for an extended time period. There is a window of safe

operation between the two extremes described above whereby electrical stimulation can occur without tissue or electrode damage. Charge-balanced or biphasic electrical stimulation avoids the problem of charge accumulation by charge balancing the positive and negative phases of the stimulation waveform. Early research in gastric electrical stimulation showed tissue "whitening" around the electrodes when low stimulation frequencies (DC to 0.5 Hz) at voltages of 5 to 8 V peak-to peak were used, possibly indicating a charge imbalance problem because of the long duration of the stimulation period (Mintchev & Bowes, 1997). In this case, the charge may have accumulated to a point where tissue damage occurred, before the charge-balanced opposite phase of stimulation was applied. Stimulation at frequencies of 5, 50 and 500 Hz was not found to produce any visible tissue damage at voltages of up to 20 V peak-to peak (Mintchev & Bowes, 1997). Charge-balance is addressed in the implant design by the precise timing of the positive and negative phases of the electrical stimulation delivered to the tissue. The frequency of stimulation will be under the control of the researcher during animal testing. The possibility of tissue damage due to charge accumulation will be monitored by the visual inspection of tissue adjacent to the electrode at the conclusion of the animal experiment, followed by histological analysis, if warranted. The possibility of electrode damage due to charge accumulation will be monitored by visual inspection of the electrode under a microscope at the conclusion of the animal experiment.

The biocompatibility of the implant relates the reactivity of the tissue to the implant surfaces. The surfaces must not react chemically nor release material to the tissue (Mayer et al., 2001). Typical materials available to encapsulate implants include some metals, ceramics, and organic materials such as epoxies, silicones, and polyurethanes.

The advantages of the organic materials are that they are conformal and that they can be applied at low temperatures. Their disadvantages are that they are not hermetic and are prone to moisture penetration in long-term use (Mayer et al., 2001), (Wise et al., 2004). The biocompatibility of the implant will be addressed in two ways: (1) cardiac pacing wires will be used for the electrodes; and (2) biocompatible silicone will be used to pot or encapsulate the implant. The biocompatible silicone will be used for rapid prototyping since it is conformal and can be applied at low temperatures.

2.3 Design Tradeoff Analysis

Based on the general block diagram shown in Figure 6, several blocks need to be defined. The following sections will address the basic design decisions required to define these blocks.

2.3.1 Power Sources for Implantable Gastrointestinal Neurostimulators

Permanent batteries are a traditional power source for implantable electrical devices. For example, Medtronic's ITREL neurostimulator has been available for the treatment of gastroparesis since 1999 (Enterra Therapy). It is an implantable neurostimulator with a permanent (i.e. non-rechargeable) battery and it implements the HI-GES stimulation method. Medtronic estimates this device's operational battery life to be 5 to 10 years, after which it will have to be surgically replaced.

Other possible power sources include transcutaneous energy transfer (TET) or a combination of TET with a secondary (i.e. rechargeable) battery as is used in heart pumps. In TET, an electrical transformer is created by using a primary (or transmitting)

coil located outside the body to transfer energy through the skin to a secondary (or receiving) coil that is implanted inside the body. When active, this receiving coil provides an AC voltage which must be filtered and rectified to power the implant circuitry.

Considering the low mortality rate for gastroparesis and constipation, the battery life for the ITREL neurostimulator seems to be insufficient to adequately treat these conditions because of the repeated surgeries required to replace the battery during the lifetime of the patient. Furthermore, the HI-GES method requires less energy than the pacing and SNGES methods. It is possible to use a much larger battery but this would be a bulky and heavy solution to extend the operational life of the implanted gastrointestinal stimulator.

The use of TET to transcutaneously power the implant was chosen to explore a batteryless design option. Collaborative research was carried out on the transcutaneously powered neurostimulator. Joint discussions with James Doherty were held to specify the interface between the implant and the belt, and it was agreed that the voltage to be delivered to the implant would be in the range of 10 to 20 V DC for implant current draws between 0 and 50 mA.

2.3.2 Voltage Regulation Methods

Since the implant can receive an input voltage in the range of 10 to 20 V, this voltage must be converted to levels suitable for the electrical devices in the implant and also for the tissue itself. A common approach to delivering the variable voltage to the tissue, as defined by Table 1, is to use a Digital-to-Analog Converter (DAC) possibly followed by an Operational Amplifier (op-amp) stage to give higher current drive

capability. It was decided that the DAC method would have been more inefficient in terms of power dissipation than simply having an adjustable voltage regulator that was controlled to produce the desired output voltage. The three basic methods for voltage regulation are: charge pump regulators, linear regulators, and switching regulators.

2.3.2.1 Charge Pump Voltage Regulators

Charge pump voltage regulators can produce voltages higher or lower than the input voltage. They also can invert the input voltage. This voltage transformation is achieved by electrically switching capacitors periodically with the input voltage, and possibly with the intermediate switched capacitor voltages, to generate the desired output voltage. Although known to be highly efficient, charge pump regulators that could accept the high input voltage range required by the implant were not found to be available commercially, so they will not be discussed further.

2.3.2.2 Linear Voltage Regulators

As shown in Figure 7, the linear voltage regulator uses a series pass transistor to control the current through the load in order to regulate the output voltage. Feedback from the output voltage, using a voltage divider, is used to control the series pass transistor.

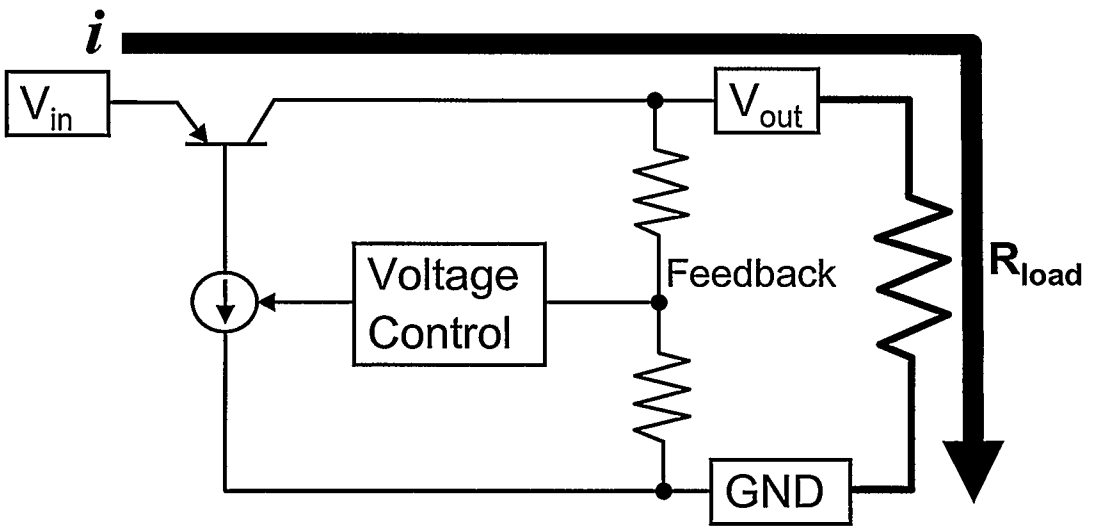


Figure 7. Linear Voltage Regulator

Linear voltage regulators are known to have very low voltage ripple on their outputs and have a very good transient response. The drawback of these regulators is that they achieve voltage regulation using power dissipation. A simple model of the power dissipation of a linear voltage regulator is:

$$P_D = (V_{in} - V_{out}) \cdot (i_{load} + i_{gnd}) \quad [1]$$

where P_D is power dissipation, V_{in} is the input voltage, V_{out} is the output voltage, i_{load} is the current through the load, and i_{gnd} is the current to ground from the voltage regulator. It can be seen that the power dissipation can be minimized if V_{in} is very close to V_{out}. In linear voltage regulators, there is a minimum required separation, called dropout voltage, between V_{out} and V_{in}. This dropout voltage depends on the magnitude of i_{load} and the type of linear voltage regulator, but can range from 100 mV to 2.2 V (Xilinx, 2003).

An example of a linear regulator which may be a good choice for the implant is the LP2951, which has a dropout voltage of 150 mV at a load current of 100 μA, and 600 mV at a load current of 100 mA. The ground current of the LP2951 is 140 μA at a load current of 100 μA, and 14 mA at a load current of 100 mA.

Some linear regulators provide external access to their feedback pins so that a user can make an adjustable voltage regulator based on the configuration of the resistive voltage divider circuit.

2.3.2.3 Switching Voltage Regulators

A switching voltage regulator uses pulse-width modulation to modulate the current flowing through an inductor to regulate the output voltage, which is filtered through an output capacitor. Relative to the input voltage, a switching regulator can be used to increase (i.e. boost), decrease (i.e. buck), or invert the output voltage.

Figure 8 shows an example of a switching buck regulator. When the series pass transistor is on, current flows through the inductor and the load. As current flows through the inductor, the inductor becomes energized and stores energy in its magnetic field. While the transistor is on, V_{out} will gradually rise towards V_{in} until an upper threshold is detected at the feedback input, whereupon the transistor will be turned off.

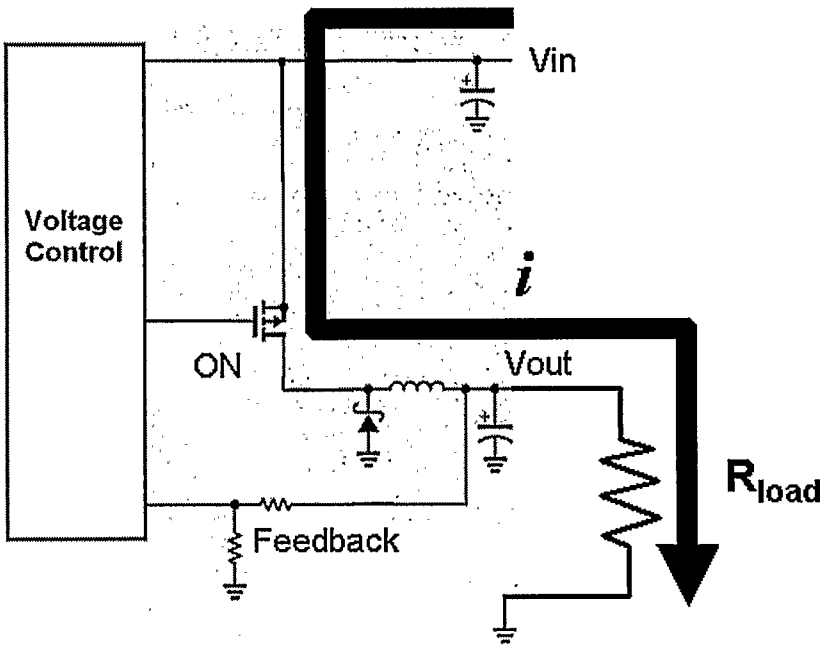


Figure 8. Step-Down Switching Regulator: Energizing the Inductor

When the transistor is off, the magnetic energy stored in the inductor will be converted to current which will drive the load until V_{out} drops below a lower threshold (detected at the feedback input), whereupon the transistor is turned back on.

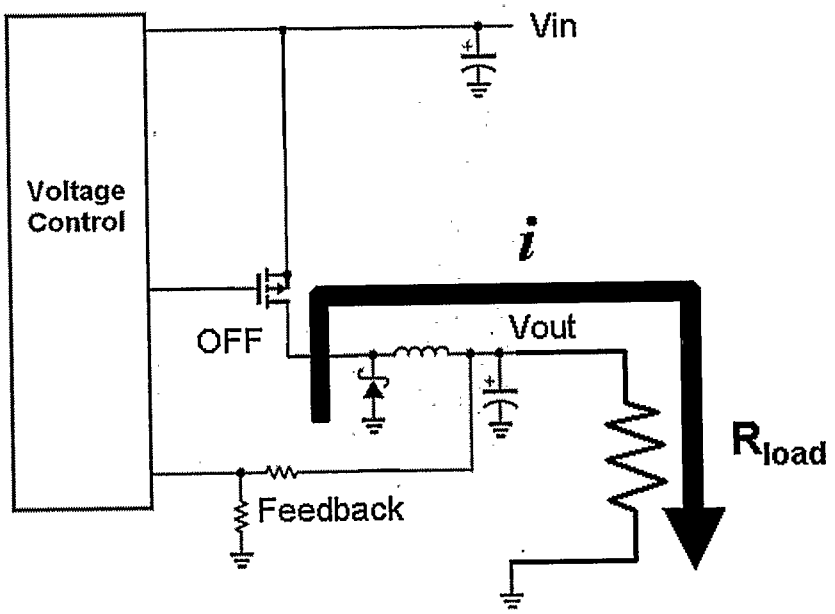


Figure 9. Step-Down Switching Regulator: De-energizing the Inductor

Switching regulators typically have more noise at their outputs than linear regulators, but are more power efficient (Xilinx, 2003). Switching regulators also are more complicated to design and require many more external components than linear regulators.

An example of a switching buck regulator that may be a good choice for the implant is the LM2671, which can have efficiencies up to 96% at 500 mA load current (National Semiconductor, 2003). The quiescent current of the LM2671 is 3.6 mA.

As with linear regulators, some switching regulators provide external access to their feedback pins so that a user can make an adjustable voltage regulator based on the configuration of the resistive voltage divider circuit.

2.3.2.4 Analysis of Voltage Regulator Options

The implant will have two main subsystems: a controller subsystem to control the electrical stimulation to the tissue and communicate with the belt; and a tissue interface circuit to deliver the electrical stimulation to the tissue.

To provide a suitable voltage to the controller subsystem, the voltage regulator must provide a voltage of 3.3 V from an input of 10 to 20 V. Normally, a switching regulator may seem an attractive option because of the large voltage drop required. However, the current consumption of the digital circuit is assumed to be quite low, below 1 mA, so the linear regulator is a suitable choice because the power consumption of the linear regulator, $(20V - 3.3V) \cdot 1ma = 16.7mW$, is negligible. Commercially available switching regulators are inefficient at low load currents and driving a load of less than 1 mA with a switching regulator may not prove to be as power efficient as the linear regulator. As an example, the LM2671 switching regulator has a quiescent current of 3.6 mA (without a load), which translates to a power dissipation of $20V \cdot 3.6mA = 72mW$,much higher than that of the linear regulator.

To evaluate the best option for the voltage regulator, both the linear and switching regulators were implemented in the prototype circuit and regulator outputs were available to stimulate the tissue. In addition, a third option was implemented by using the switching

regulator as a pre-regulator to the linear regulator to achieve high efficiency and clean output voltages. The results of this comparison are shown in Chapter 4.

2.3.3 System Controller

The system controller will be the part of the implant circuit that sends and receives data to and from the belt using the modulator and demodulator circuit. Based on the desired stimulation pattern received in the data from the belt, the system controller will control the signals to the tissue interface circuit that delivers the electrical stimulation to the tissue. The balanced timing of these control signals is critical to prevent a charge imbalance in the stimulated tissue. From the requirements of charge balance and the wide variety of stimulation patterns, a digital circuit in the form of a microcontroller or an Application Specific Integrated Circuit (ASIC) or an ASIC-like device was decided to be the appropriate choice for a system controller.

Since minimization of power dissipation is an important parameter in the design of the implant, a method to minimize the power consumption of digital circuits was explored.

A simple model describes the dynamic power dissipation of a CMOS digital circuit as:

$$P_D \propto C \cdot V^2 \cdot f \quad [2]$$

where P_D is the dynamic power dissipation, C is internal switched capacitance of the circuit, V is the supply voltage, and f is the clock frequency. For static power dissipation, the relation is:

$$P_S = I_Q \cdot V \quad [3]$$

where P_S is the static power dissipation and I_Q is the quiescent current. Since the clock frequency is under designer control, it will be used as the parameter to judge the relative power efficiency of the alternative options. Another parameter that will be considered is the relative ease of designing with the selected option for an implant system controller.

2.3.3.1 Microcontroller Option

A microcontroller is a device that is well suited for embedded applications. It integrates a microprocessor with common peripheral devices such as memory, timers, and Universal Asynchronous Receiver/Transmitter's (UART's) within one package. For low-power applications, a common practice is to use a special microcontroller function to put the microcontroller to "sleep" when its operation is not required. When a microcontroller is in sleep mode, most (if not all) of its clocks are shut off to conserve power. Typically, a microcontroller can be brought out of sleep mode with an interrupt from a timer or an external source. A microcontroller that may be a suitable system controller for the implant is the Atmel ATtiny2313 (Atmel 2004). At a clock frequency of 4 MHz and a supply voltage of 3 V, an active ATtiny2314's current consumption is 2mA. In sleep mode with the timers running, the current consumption is 1mA. In sleep mode with all the clocks off, the current consumption is 2 μA. This figure can be taken to be the quiescent current of the microcontroller.

Meeting the charge balance and timing requirements with a microcontroller controlling the electrical stimulus to the tissue can be problematic at low clock frequencies due to interrupt latency and the sequential processing of program

instructions. To obtain the precise timing required to generate the electrical stimulus to the tissue, it would be desirable to use the microcontroller's timer peripheral to generate an interrupt at the correct time to turn an electrical stimulus either on or off for any one of the four output channels to the tissue. When the interrupt is generated, the microcontroller must perform a context switch to be able to service the interrupt. In this context switch, the current state of the system is stored and the time that it takes is called the interrupt latency. For the ATtiny2313, the interrupt latency is at least 7 clock cycles when the microcontroller is active or at least 11 clock cycles when the microcontroller is in sleep mode. It takes 4 clock cycles to return from the interrupt. When the timer-driven interrupt handler or interrupt service routine (ISR) runs, a set of sequential statements will be executed to reset the timer (4 clock cycles) and perform any other functions. The control signals to turn electrical stimulation on or off to any one of the four output channels can be generated in the ISR or the main body of the program. One can determine the minimum possible time between timing events in the electrical stimulation to the tissue. From Table 1, the highest required frequency is 250 Hz and the minimum required duty cycle is 10%. As shown in Figure 10, the time interval between the rising and falling edges of a stimulation pulse (in the lower waveform) is the minimum time between events that the microcontroller has to react, in this case 0.4 ms. Assuming that a timer interrupt controls the timing for Edge 1 and Edge 2, then taking into account only the interrupt latency (7 clock cycles), resetting the timer (4 clock cycles), and the return from interrupt (4 clock cycles) and ignoring any sequential statements in the ISR, at least 15 (i.e. 7 + 4 + 4) clock cycles must occur to meet the 0.4 ms timing for the pulse.

Therefore, the absolute minimum clock frequency to the microcontroller must be $\frac{15}{0.4ms} = 37.5kHz$.

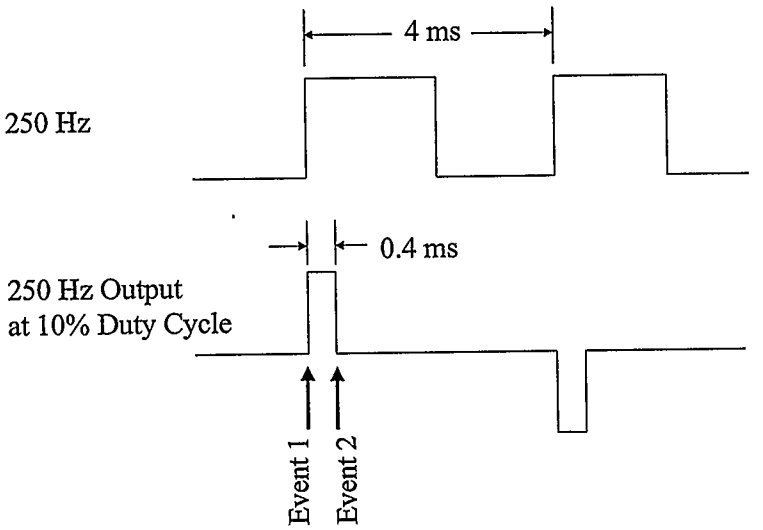


Figure 10. Minimum Timing Interval Between Events

Developing code for a microcontroller is relatively easy compared to designing an ASIC. Microcontroller peripherals (memory, timers, UART's, etc.) are pre-designed and integrated with a microprocessor on the same die. Access to these peripherals is through easy to use registers. Design tools such as "C" compilers, libraries, and operating systems shield the program designer from many of the low-level details of the microcontroller specific device architecture, thus enabling rapid development. Another advantage of microcontrollers is that they are typically reprogrammable, enabling programs to be tested through many design iterations.

2.3.3.2 ASIC Option

A digital ASIC is a custom, specialized circuit to serve a particular purpose. It is on the other end of the design spectrum from a microprocessor, which can be programmed for many different applications. Although microprocessors can be embedded in ASIC's, only custom digital logic will be considered in this analysis because the microcontroller option discussed in the previous section also applies to microprocessors embedded in ASIC's.

ASICs can implement combinatorial or sequential circuits, or both. One of the powerful features of ASICs is that they can also implement parallel circuits to perform various functions. Parallelism offers the ASIC the ability to execute operations concurrently and in the case of the timing shown in Figure 10, no interrupt latency is required, as in the case for the microcontroller. Therefore, to meet the timing described in Figure 10, the ASIC would have to be clocked at only $\frac{1}{0.4ms} = 2.5kHz$, which is 15 times slower than the microcontroller.

ASICs are customized circuits that perform a specialized function and are not adaptable to modification once they are made. ASICs are costly and time-consuming to manufacture. It takes several months to obtain physical samples of an ASIC design that has been submitted for manufacture. Accordingly, much design effort and simulation must go into designing an ASIC before submitting it for manufacture. Typically, an ASIC circuit is designed using a Hardware Description Language (HDL) such as Verilog or VHDL. The design effort for implementing a function in an HDL is much higher than writing a program for a microcontroller/microprocessor to implement the same

functionality. Once an ASIC is manufactured, it is almost impossible to salvage it if there is a design or manufacturing defect.

2.3.3.3 FPGA/CPLD Option

An alternative to an ASIC is a Field Programmable Gate Array (FPGA). Essentially, an FPGA is like a programmable ASIC. The design entry method uses an HDL but unlike an ASIC, an FPGA can be repeatedly reprogrammed. Physically downloading a configuration bitstream to an FPGA is as simple as downloading a new program to a microcontroller, thus enabling many design iterations during development. In fact, it is a common practice to use FPGAs to prove the functionality of HDL code before manufacturing an ASIC.

For this application, the timing for an FPGA would be the same as it would for the ASIC, with a minimum clock frequency of 2.5 kHz. For power consumption, a very low power alternative is available to the FPGA, the CoolRunner Complex Programmable Logic Device (CPLD), manufactured by Xilinx Incorporated. A CPLD can implement combinatorial and sequential logic like ASICs and FPGAs, is programmable like an FPGA, and is non-volatile like an ASIC.

2.3.3.4 Analysis of System Controller Options

Taking into consideration the very low operating frequencies of the microcontroller, ASIC, CPLD options, the expected power dissipation would be very low in any of the cases. The microcontroller current consumption at 32.5 kHz would be below 100 μA (Atmel, 2003). The CPLD current consumption at 2.5 kHz would be approximately 25 μA (Xilinx, 2004b). The power dissipation for the microcontroller and

voltage regulator would be $(20V-3.3V)\cdot(100\mu A+140\mu A)=4mW$, and for the CPLD would be $(20V-3.3V)\cdot(25\mu A+140\mu A)=2.8mW$, where 140 μA is the quiescent current of the voltage regulator. Both these values are relatively small compared to the tissue power dissipation of 25.4 mW per channel (colonic example).

From the preceding clock frequency analysis, it would be reasonable to expect a lower relative value in power consumption for the CPLD, in comparison to the microcontroller. At such low frequencies, however, the quiescent currents of the linear regulator and the CPLD/microcontroller dominate the power consumption.

In the end, the CPLD option was chosen to facilitate the testing of HDL code on a prototype before proceeding with an ASIC implementation in a future iteration of the implant design. This future ASIC implementation would incorporate almost all of the digital and analog components of the implant, minimizing the size and power consumption of the circuit.

2.3.4 Data Transmission Methods

2.3.4.1 Physical layer

The data transmission method from the belt to the implant is Amplitude Modulation (AM) of the transcutaneous power link. To send data from the implant to the belt, a Load-Shift-Keying (LSK) technique is used. In LSK, large-valued resistive load is shifted in and out of the circuit in order to change the reflected impedance of the implant. This change in reflected impedance can be detected at the belt.

2.3.4.2 Data Integrity

Two common methods to ensure data integrity are to use Forward Error Correction (FEC) or Automatic Repeat Request (ARQ) (Ramabadran & Gaitonde, 1988). FEC typically is used for data transmission cases where retransmission is not an option, e.g. streaming video. In ARQ, the received message is checked for error detection, and if there are errors, a request is sent for the data to be repeated. The ARQ method is appropriate for the implant because very small amounts of data are transmitted and there is no practical limit on how many times a message can be retransmitted until the data is received without errors.

A Cyclic Redundancy Code (CRC) will be appended to the data packet to ensure its data integrity using the ARQ method. The binary data word is treated as a polynomial, which is divided by a CRC generator polynomial (Koopman, 2002). The remainder of this division is the CRC code which is appended to the data packet for transmission. When the data packet is received, the CRC code is computed using the data packet and compared with the received CRC code. If the CRCs are the same, the data is processed. If they are different, then a repeat request gets sent. A CRC is simple to implement in hardware, using a Linear Feedback Shift Register (LFSR) (Ramabadran et al., 1988), with each CRC bit requiring one flip-flop.

CRCs come in a variety of bit sizes. Commonly used are 8-, 16-, and 32-bit CRCs, although CRCs of any size can be generated. Generally, the higher the number of bits in the CRC, the greater is the error detection power of the CRC. CRC codes are not fool-proof; there is a possibility that a message containing errors can pass the CRC check. A measure of the power of the CRC to detect errors is given by the Hamming distance

(HD) (Koopman & Chakravarty, 2004). The HD defines the minimum number of bit errors that must be in a message for that erroneous message to be undetectable by the CRC. The HD depends on the particular CRC generator polynomial and the data word length. The data length of the packet to the implant will be between 128 and 256 bits and the 32-bit CRC polynomial that has been proven to have the largest HD for data words up to 274 bits, 0xFA567D89, was found by Castagnoli, Braeuer, & Herrman (1993), with an HD of 8 bits (Koopman et al, 2004). The Castagnoli polynomial will be used in the CRC check.

2.3.4.3 Transmission Protocol

For simplicity, the data will be transmitted in the UART protocol, which is shown in Figure 11. Since the UART protocol is asynchronous, the timing between the receiver and the transmitter must be established. The receiver must be configured in advance to the same data rate as the transmitter. To begin a transmission, a UART sends a start bit to enable the receiver to begin timing the reception of the data bits. The data bits are sent 8 bits at a time, from Least-Significant-Bit (LSB) to Most-Significant-Bit (MSB). The transmission is terminated with either 1or 2 stop bits.

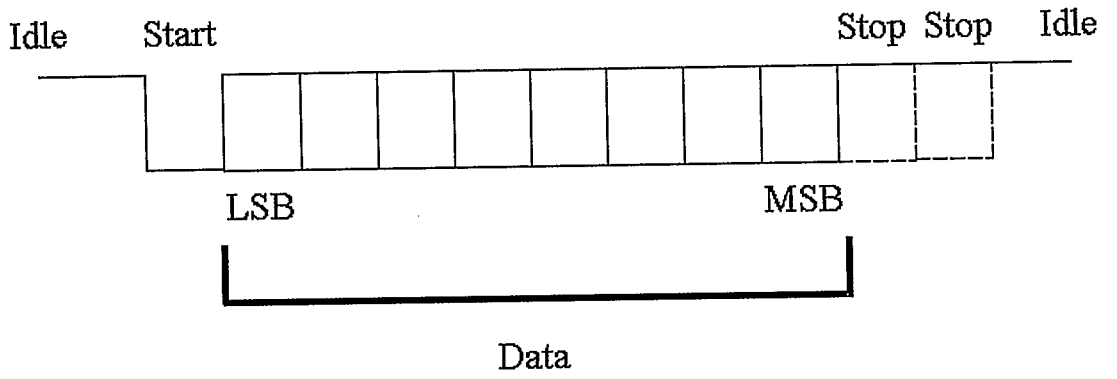


Figure 11. UART Protocol

The transcutaneous communications link between the belt and the implant was not able to tolerate long periods of zeros in a data byte so Manchester encoding was implemented with the UART protocol. This was accomplished by recoding every data bit into two bits, e.g. a 1 becomes 10 and a 0 becomes 01. The Manchester protocol ensures a signal level transition for each data bit, but at the expense of doubling the required bandwidth. Figure 12 shows an example of Manchester encoded data. The data byte, 11011000, is first transformed to two Manchester encoded data bytes, 10100110 and 10010101. The Manchester encoded data bytes get sent LSB first as 01100101 and 10101001. A further benefit of Manchester encoding is that it provides a degree of error checking in that no data bit can be comprised of two ones or zeros. This feature provides protection against single bit errors and more importantly against burst errors for which the CRC code can be relatively weak. Thus both the CRC and Manchester codes work in a complimentary fashion with each other.

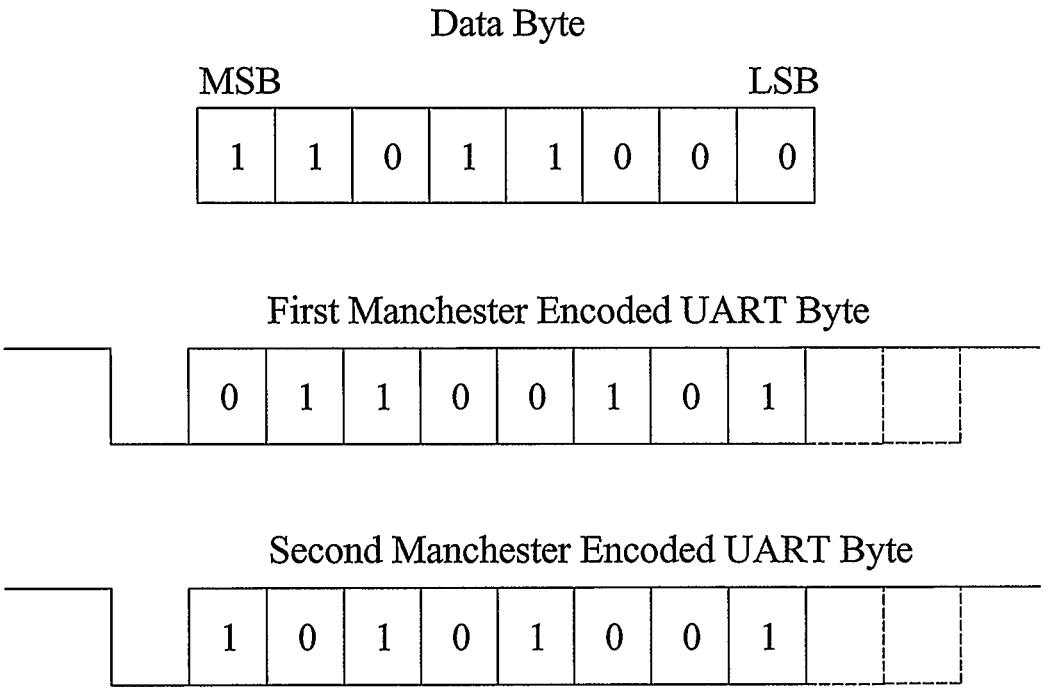


Figure 12. Example of Manchester Encoded Data

2.4 Overview of the Xilinx CoolRunner-II XC2C512 CPLD

The CPLD chosen for the implant design is the Xilinx CoolRunner-II XC2C512 (Xilinx, 2004b). This CPLD is the largest CPLD available, with 12,000 equivalent ASIC gates, approaching the size of smaller FPGA's. An ASIC gate is defined by the area taken up by a 2-input NAND gate and is a common measure of the size of a design, or the area available for a design. Since a CPLD is not an ASIC, the term "equivalent gates" is used in order not to confuse it with the actual area on the CPLD. The programmability features of a CPLD cause it to take up much more area (e.g. about 10 times) than an ASIC for any given application. This means that there is a price paid for programmability feature of a CPLD, in terms of both area and power consumption.

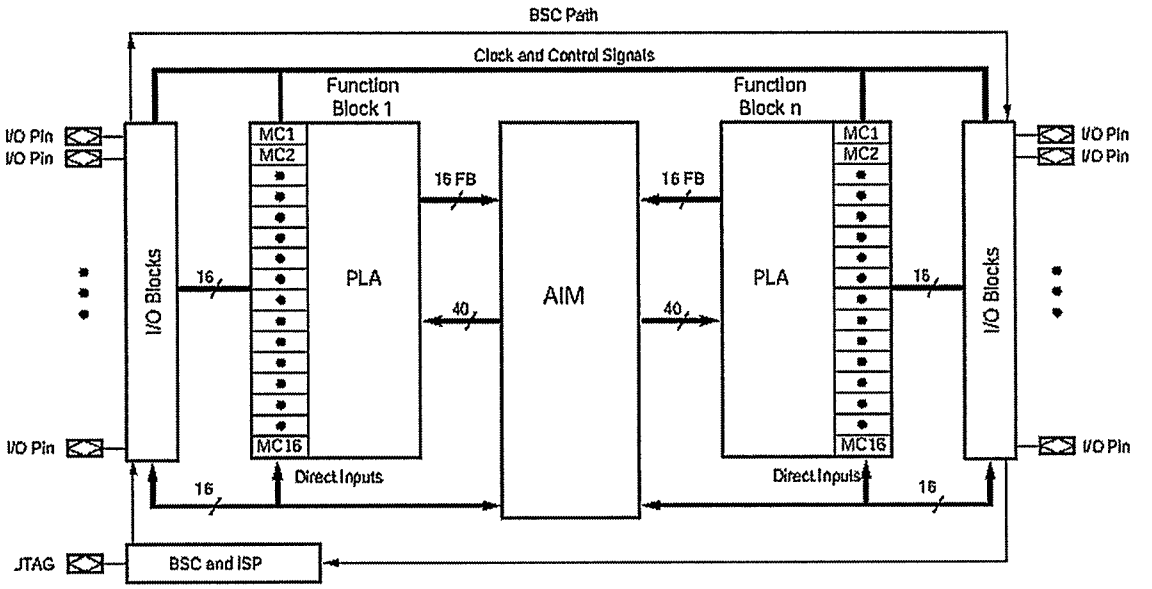


Figure 13. CoolRunner-II CPLD Architecture (Xilinx, 2004a)

The Xilinx XC2C512 has 32 function blocks connected to each other by the Advanced Interconnect Matrix (AIM), as shown in Figure 13. Each function block receives 40 true and inverted inputs from the AIM. The function blocks are made up of a Programmable Logic Array (PLA) and 16 macrocells, as shown in Figure 14. The 16 outputs of the macrocells go to the AIM, to be re-routed to other function blocks, or to the output pin buffers of the CPLD. The AIM, macrocells, and PLA are described in further detail in the next paragraphs. The Input/Output (I/O) blocks are the electrical interface between the I/O pins, which connect to the circuitry outside the CPLD, and the internal logic of the CPLD. The JTAG I/O pins are the CPLD interface for In-System Programming (ISP), whereby the CPLD logic can be configured with a particular logic implementation. The JTAG interface also supports the Boundary Scan Chain (BSC),

which supports the external reading of the logic implementation of the CPLD, to ensure that the implementation was successfully configured.

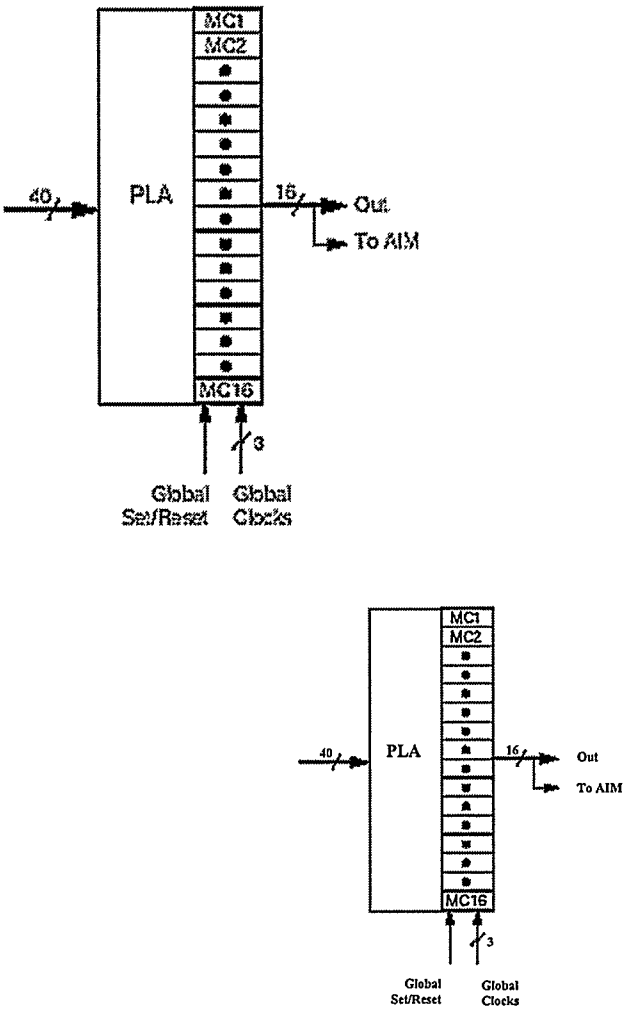


Figure 14. CoolRunner-II CPLD Function Block (Xilinx, 2004a)

A detailed view of the PLA is shown in Figure 15. A set of logical AND arrays receives 40 true or inverted inputs from the AIM. Each AND array is called a product term and has 80 inputs. Schematically, the connections in the diagram are shown as

circles and no connections are shown as "X". The product terms then are fed into a logical OR array whose outputs go towards the macrocell outputs.

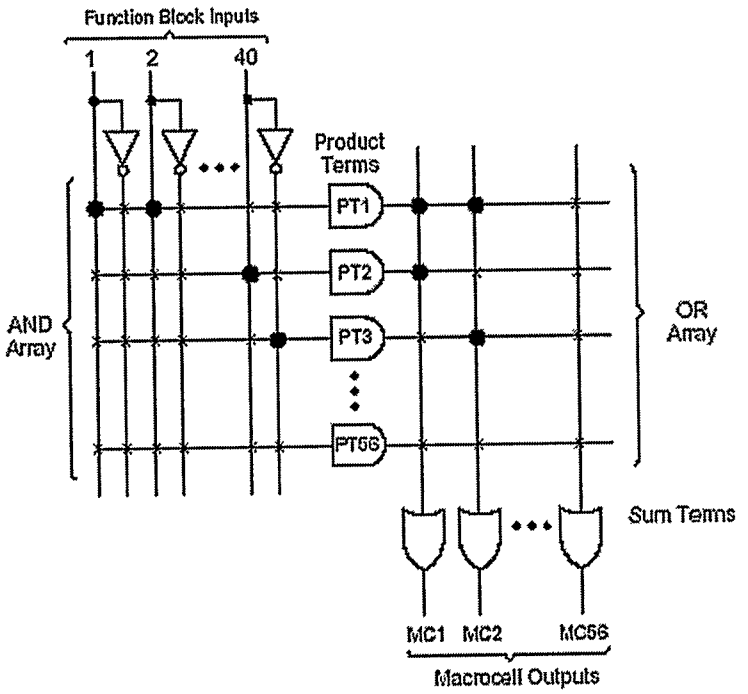


Figure 15. Basic PLA Architecture (Xilinx, 2002)

A diagram of a macrocell is shown in Figure 16. The output of the PLA OR term is fed to an XOR gate for conditional inversion, if required. The XOR output can be fed directly to the output buffer or it can be multiplexed to be an input to a flip-flop or latch.

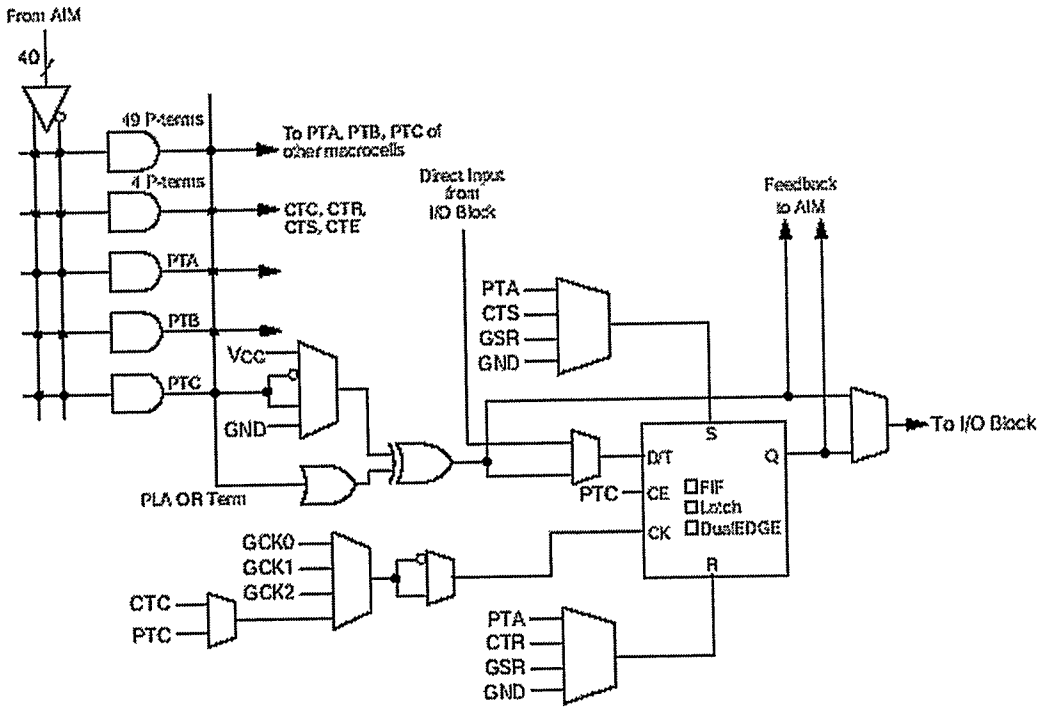


Figure 16. CoolRunner-II CPLD Macrocell (Xilinx, 2004a)

This CoolRunner-II CPLD is designed for low-power battery-based applications and has a variety of low-power enabling features, such as CoolCLOCK and DataGATE. These special features were not used in the implant design to preserve the portability of the design to an ASIC.

The Xilinx Integrated Software Environment (ISE) 6.2i was used as the design entry, synthesis, and programming tool for the CPLD. ModelSim XE/Starter 5.5e_p1 was used as the simulation tool to verify the behaviour of the design before downloading it to the prototype.

2.5 Overview of Verilog Programming

Verilog is an HDL that was developed in the early 1980's by Gateway Design Automation (Smith, 1996). An HDL is a programming language used to model digital hardware. HDLs such as VHDL and Verilog evolved to increase the productivity of designers by allowing them to design circuits in a more abstract fashion than is possible using traditional schematic entry techniques. An important distinction to make between software programming languages and HDLs is that an HDL is a *description* language so, in essence, it *describes* the circuit being modelled, as a form of documentation. Another important distinction is that HDLs model parallel behaviour in time, rather than the sequential order of statements executed in a software programming language. This parallel feature allows the modelling of parallel circuits operating concurrently, exactly as in real circuits. Designs in Verilog are written on a module by module basis. These modules describe circuits and can be instantiated within other modules on a hierarchical basis.

To help ensure the quality of the Verilog code and design approach, the *Reuse Methodology Manual* (Keating & Bricaud, 1999) was as used as guidance. In particular, the Register Transfer Level (RTL) coding guidelines were followed closely. At the RTL level, the circuit is described by a subset of the HDL that is synthesizable (i.e. can be translated to a physical circuit) and which corresponds to the data flow of signals between registers (or flip flops). Non-synthesizable HDL can be useful for the implementation of testbenches to verify the functionality of the RTL design in simulation. Almost all of the Verilog modules developed for the CPLD were first tested in simulation with Verilog testbenches.

CHAPTER THREE: DETAILED IMPLANT DESIGN

This chapter describes the details of the design implementation of the implant. It discusses the major components and the details of the Verilog implementation in the CPLD.

3.1 Detailed Block Diagram of the Neurostimulator System

From the preceding chapter, a detailed block diagram of the neurostimulator system has been developed as shown in Figure 17.

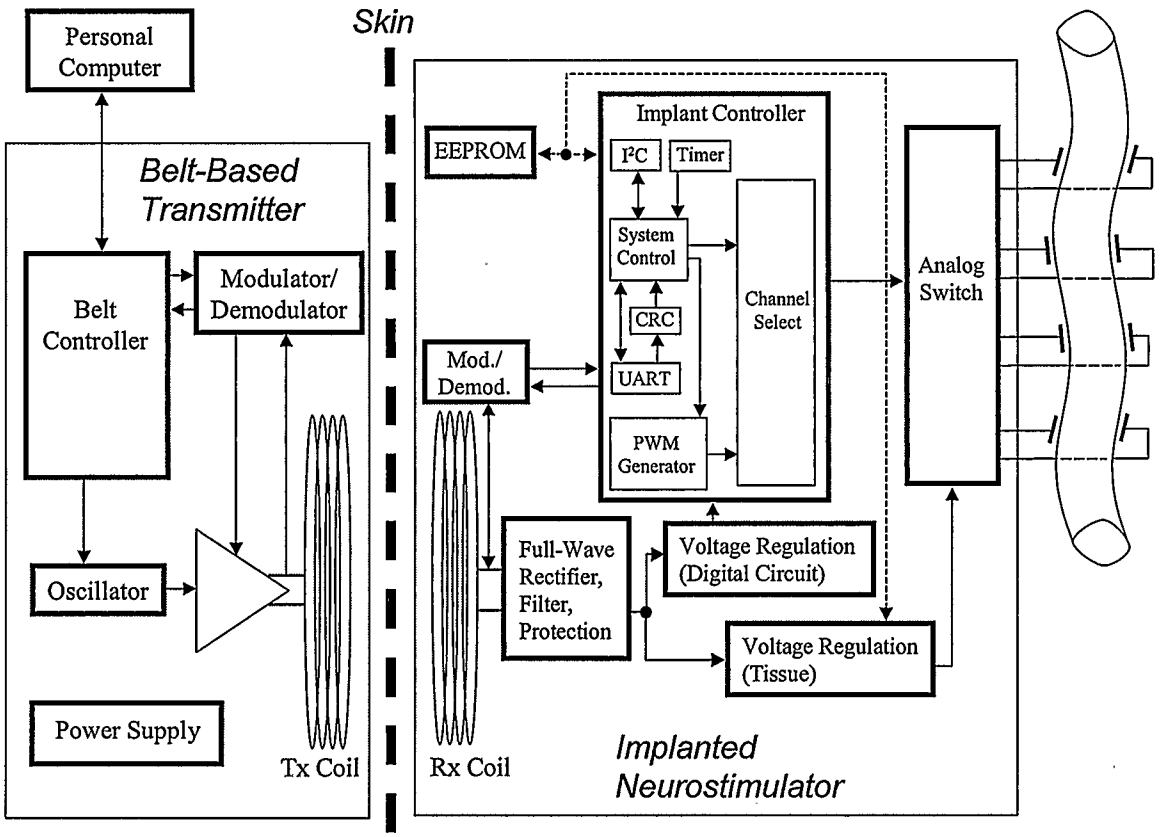


Figure 17. Detailed Block Diagram of Neurostimulator System (Colonic Example)

The belt contains:

4. a transmitting coil and a power amplifier which drive the transmitting coil in order to provide energy to the implant;
5. an oscillator which generates the correct frequency to modulate the power amplifier for optimum energy transfer;
6. an AM modulator which modulates the amplitude of the transmitted energy to the implant;
7. a demodulator (envelope detector) which receives LSK data from the implant;
8. a power supply (battery) which provides energy to the belt circuitry;
9. a controller which:
    d. transmits configuration data to the implant using the modulator;
    e. receives status data from the implant using the demodulator;
    f. communicates with a personal computer to receive configuration data to be sent to the implant and to send status information to the computer from the implant;
    g. enables the oscillator.

The implant contains:

5. a receiving coil to transcutaneously obtain energy for the implant;
6. a rectification and filtering circuit to provide DC voltage to power the rest the implant;

7. a linear regulator to convert the DC input voltage to power the digital electronics;
8. a modulator that switches a load resistor to transmit data to the belt via LSK modulation;
9. a demodulator (envelope detector) to receive AM data from the belt;
10. an adjustable linear and adjustable switching regulator to efficiently provide the correct voltage to the tissue
11. an analog switch circuit to drive the electrical stimulation signals to the tissue;
12. a serial EEPROM to store the particular configuration data for the subject;
13. a CPLD to function as the implant controller to:
    d. receive configuration data from the belt using the demodulator;
    e. send status data to the belt using the modulator;
    f. generate the control signals to the tissue interface circuit.

My collaborator, James Doherty, worked on: the belt circuitry; the transmitting and receiving coil design; and the modulator/demodulator circuit. The remainder of Chapter 3 discusses my work that consisted of the rest of the implant design, including: the implant controller; the voltage regulation circuits; and the analog switch circuitry.

3.2 Clock Source

Although the theoretical lower limit for the system clock is 2.5 kHz, as discussed in Chapter 2, it was not a practical frequency to obtain commercially. The lowest

frequency clock oscillator found operated at 32 kHz, but a 32.768 kHz clock oscillator was chosen instead because it could easily be divided to generate a 1 Hz time base.

3.3 CPLD Implementation

3.3.1 CPLD Design Architecture

The design of the CPLD is implemented in Verilog and has 8 main modules connected together by a top-level module. The block diagram, signals, and interfaces are shown in Figure 18. The complete listing of the Verilog code is given in Appendix A. The dashed signal lines indicate strobe signals that, when active, only remain active for one clock cycle.

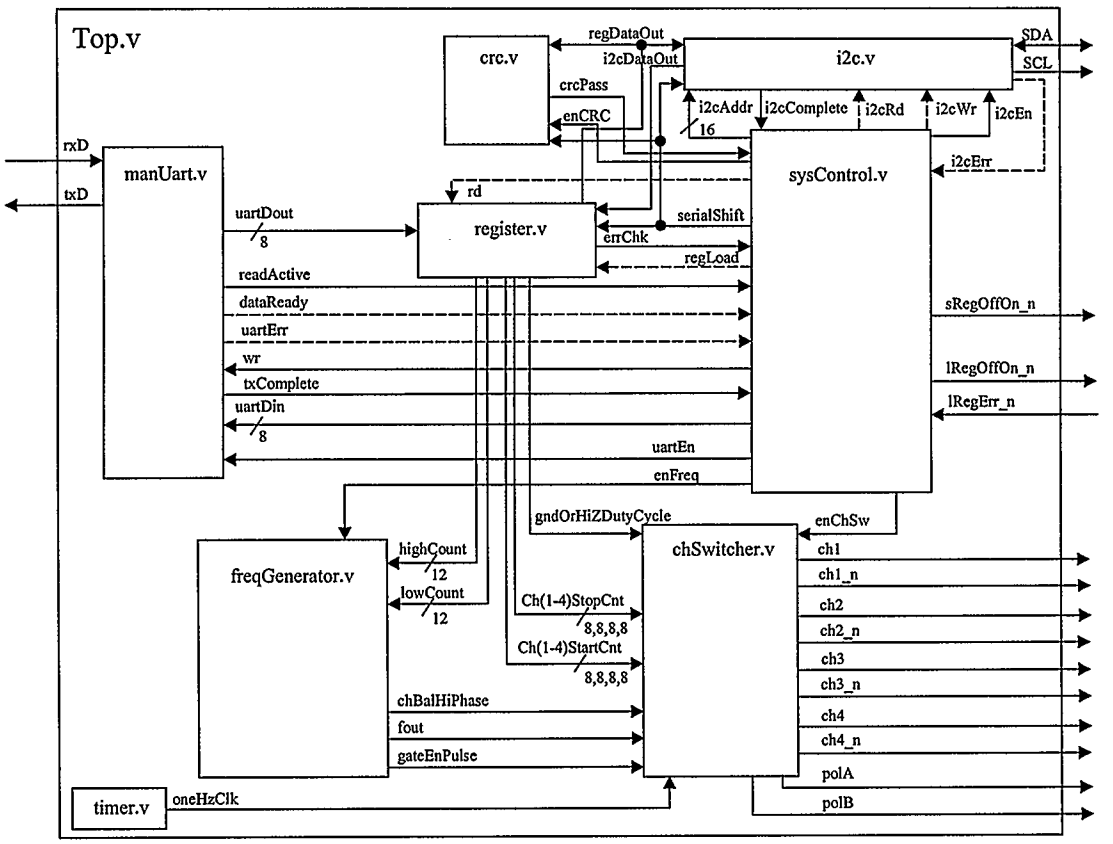


Figure 18. Implant CPLD Design Architecture

The individual blocks of Figure 18 are described in detail in the following sub-sections, however a brief description of the block diagram is presented in this paragraph. The *sysControl* block functions as the controller for the CPLD. It receives error messages and status signals from the other blocks in the CPLD and from the rest of the implant circuit. The *sysControl* block controls the other blocks in the CPLD and the implant circuit in a deterministic sequence of operations that facilitate the goal of stimulating the gastrointestinal tissue. The *register* block holds the configuration information for the stimulation pattern. The *freqGenerator* and *chSwitcher* blocks generate the signal to the

tissue based on the *register* block data. The *register* block contents are provided by data coming from belt, through the *manUart*, or from the serial EEPROM on the implant circuit. The *i2c* block is the serial interface controller to the serial EEPROM and also digital potentiometers on the implant circuit. The *crc* block provides a data integrity check to data coming in to the *manUart*.

The resource usage of the CPLD for the implant design is given in Table 2. As can be seen, the CPLD has a high degree of utilization by the code. The degree of usage precludes migrating this design to a smaller Xilinx CPLD.

Table 2. CPLD Resource Usage for Complete Design

Module	Macrocells (total 512)		Pterms (total 1792)		Registers (total 512)	
Name	Used	%	Used	%	Used	%
top.v	436	86	1302	73	362	71

Table 3 shows the utilization of the sub-modules that compose the design. If added together, the usage numbers exceed the numbers given in Table 2. This is because resource sharing and optimization during the synthesis of the top-level design leads to a more efficient implementation.

Table 3. CPLD Resource Usage for Individual Modules

Module Name	Macrocells (total 512)		Pterms (total 1792)		Registers (total 512)	
	Used	%	Used	%	Used	%
resetBlock.v	5	1	5	1	5	1
register.v	144	29	298	17	144	29
manUart.v	60	12	175	10	41	9
i2c.v	39	8	166	10	28	6
freqGenerator.v	40	8	107	6	18	4
crc.v	34	7	86	5	32	7
chSwitcher.v	38	8	177	10	29	6
sysControl.v	83	17	328	19	52	11

3.3.1.1 CPLD Controller – Module sysControl.v

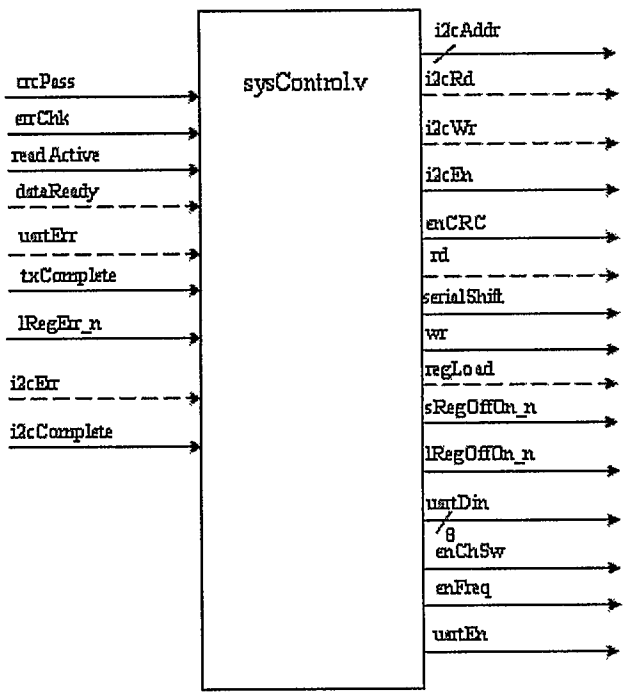


Figure 19. Block Diagram of sysControl.v

This module is the main controller for the CPLD. It has a 19-state state machine to control the flow of operations during a stimulation session.

Upon power-up from the belt, sysControl waits 30 seconds to allow the power to stabilize. It then waits for 90 seconds for a configuration message from the belt. This configuration message contains the desired stimulation parameters. If no message is received, then the serial EEPROM is read to obtain stored stimulation parameters. Then, the voltage regulators are turned on and configured to the correct output voltages and the module that controls the stimulation to the tissue is enabled. When the stimulation is complete, a message is sent the belt.

If a configuration message is received for the belt, then this message is checked to determine if it is complete and if it passes the CRC check. If it fails on either account, then a message is sent back to the belt and no stimulation takes place. If it passes, then the serial EEPROM is programmed with the new configuration data and the tissue stimulation procedure begins.

Anytime an error is detected, sysControl either stops stimulating or does not begin stimulating and immediately sends a message to the belt indicating the type of error encountered. These messages repeat every second as long as there is power to the implant. The list of messages and their meanings is given in Table 4.

Table 4. Implant Messages to Belt

Binary Code (Manchester Encoded)	Mnemonic	Comment
0101_0101	uartError	stop bit not detected at the correct time from the UART data transfer
0101_0110	crcError	CRC check failed
0101_1001	i2cError	acknowledgement failure from a device on the I2C bus
0101_1010	pwmCompleteUART	tissue stimulation complete with parameters received from UART
0110_0101	vRegError	voltage regulator is out of regulation, indicating insufficient input voltage to the adjustable regulator (possibly from a short between electrodes or insufficient energy being transferred from the belt to meet the stimulation parameters
1001_1001	sysContErr	the state machine in sysControl goes into an illegal state (this is a trap to guard against any unforeseen glitch that may cause undesirable behaviour)
1001_1010	incompleteUartErr	insufficient no. bytes received before 30 second timeout
1010_0101	manError	Manchester encoding failure on received byte from UART
1010_0110	demodError	idle state for the receive data signal to the UART is 0, indicating an error

Binary Code (Manchester Encoded)	Mnemonic	Comment
		with the envelope detector (mostly useful for debugging)
1010_1001	pwmCompleteSprom	tissue stimulation complete with parameters received from serial EEPROM

3.3.1.2 UART – Module manUart.v

The UART is a module with a 9-state state machine. The UART is capable of full-duplex operation, but normal usage in the implant application is half-duplex. The UART performs Manchester decoding on the received data bytes. Since Manchester encoding doubles the number of data bits sent, two bytes must be received for the UART to deliver one byte to the system.

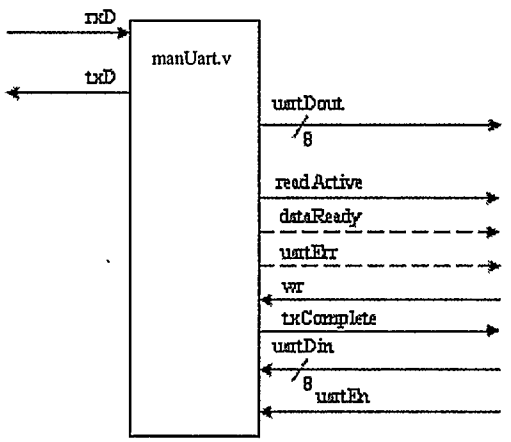


Figure 20. Block Diagram of manUart.v

The UART samples the data three times at the midpoint of the data bit to provide some noise immunity when receiving data. The space between samples is one clock period of the 32.768kHz system clock. Majority voting determines the value of the bit.

The UART detects a variety of errors in the reception of data as shown in Table 4. It detects Manchester errors, stop bit errors, and level errors when there is no signal being transmitted.

The UART does not perform Manchester encoding when transmitting data. In this case, the data itself must be coded in Manchester format. Practically, this is accomplished by pre-coding the messages to be sent in Manchester format, as shown in Table 4.

The UART operates at a baud rate of 2400 bits-per-second (bps). To derive this frequency from the 32.768 kHz system clock, one must count $\frac{32768}{2400} = 13.653$ system clock cycles in order to obtain the 2400 Hz UART clock period. The 2400 Hz clock frequency is generated by a circuit within the UART that repeatedly counts 14 system clock cycles twice and 13 system clock cycles once. This yields an average of $\frac{14+14+13}{3} = 13.666$ system clock cycles per 2400 Hz period or a UART clock rate of $\frac{32768}{13.666} = 2398$ Hz.

3.3.1.3 Module register.v

The register module is a serial shift register with parallel and serial loads. It can be configured as a ring, which means that its serial output is connected to its serial input. It can accept an 8-bit parallel load from the UART and a serial input from the I^2C module.

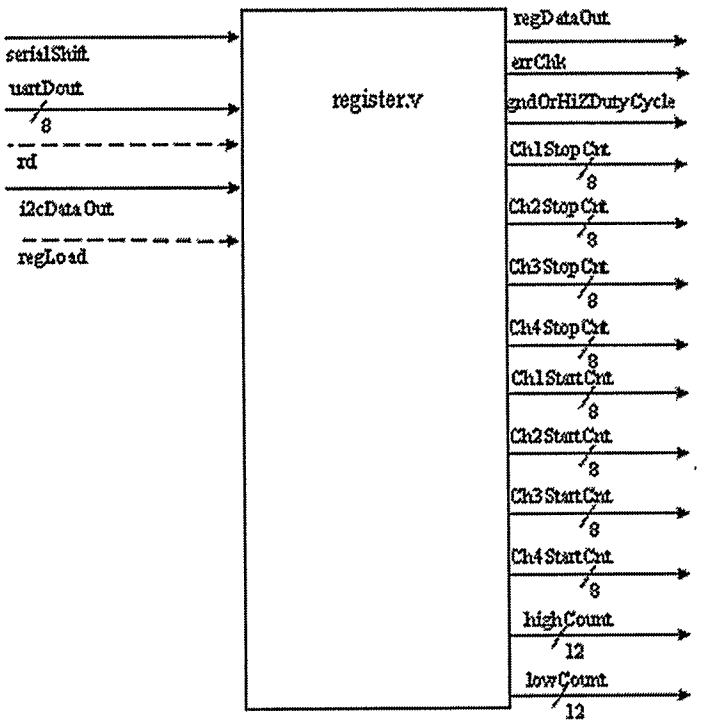


Figure 21. Block Diagram of register.v

The register module stores the configuration data for the stimulation pattern to be imposed on the tissue. It also stores the CRC that is appended to the data packet received by the UART. Since flip-flops power up in an unknown state, the register module must be loaded with data either from the UART or from the serial EEPROM before being used. The configuration of the register module is shown in Table 5.

Table 5. Register Map of CPLD

Register Bits	No. Bits	Name	Comment
143:136	8	lRegVal	linear regulator output value control
135:128	8	sRegVal	switching regulator output value control
127:116	12	highCount	no. clock cycles in high phase of frequency
115:104	12	lowCount	no. clock cycles in low phase of frequency
103	1	gndOrHiZDutyCycle	grounded (0) or high impedance (1) outputs when in low phase of output duty cycle
102:95	8	ch1StartCnt	no. seconds before channel 1 starts stim.
94:87	8	ch1StopCnt	no. seconds until channel 1 ends stim.
86:79	8	ch2StartCnt	no. seconds before channel 2 starts stim.
78:71	8	ch2StopCnt	no. seconds until channel 2 ends stim.
70:63	8	ch3StartCnt	no. seconds before channel 3 starts stim.
62:55	8	ch3StopCnt	no. seconds until channel 3 ends stim.
54:47	8	ch4StartCnt	no. seconds before channel 4 starts stim.
46:39	8	ch4StopCnt	no. seconds until channel 4 ends stim.
38:7	32	CRC	the 32-bit CRC received from the UART
6:0	7	spare	unused

3.3.1.4 Module crc.c

The CRC module is a simple 32-bit LSFR that implements the 32-bit 0xFA567D89 CRC polynomial, as described in Chapter 2. When it is enabled, it serially clocks in the data to compute the CRC. It processes the entire 144-bit register and generates a pass or fail signal that is valid when the last bit is processed.

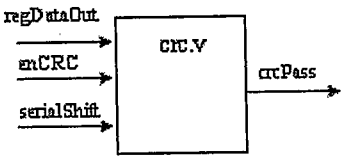


Figure 22. Block Diagram of crc.v

3.3.1.5 I^2C Controller – Module i2c.v

The Inter-Integrated Circuit (I^2C) was invented by Phillips (2004) and is a common serial interface used in embedded applications. It is a 2-wire synchronous bus that allows one or more master devices to control a large number of slave devices. The SDA pin is the bi-directional synchronous data pin and the SCL pin is the synchronous clock generated by an I^2C master device. The I^2C module operates as a master device to control two digital potentiometers and one serial EEPROM. This module is a complicated design with a 19-state state machine.

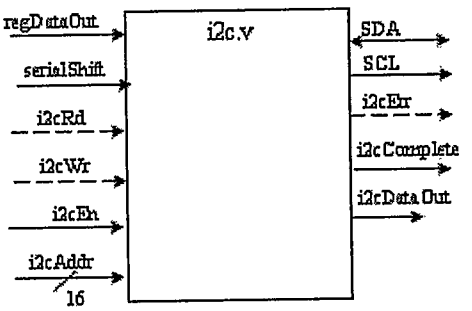


Figure 23. Block Diagram of i2c.v

The I^2C module accepts data serially and accepts the slave device address and the optional slave control word in parallel. The protocol is illustrated in the figures below. In Figure 24, the I^2C protocol is shown for the digital potentiometer. The top two sets of waveforms show a write operation to the digital potentiometer. The first byte transferred is the address followed by an R/W bit, which is low to indicate to the digital potentiometer that it is a write transfer. The second byte is an instruction byte to the digital potentiometer and the third byte is the data byte to be written to the digital potentiometer. Each of these bytes are followed by an acknowledge bit that is asserted by the digital potentiometer, following the I^2C slave protocol. The I^2C master checks that the acknowledge bit is asserted to ensure that the device is operating correctly. If there is an acknowledgement error, then an error signal is asserted to the sysControl module.

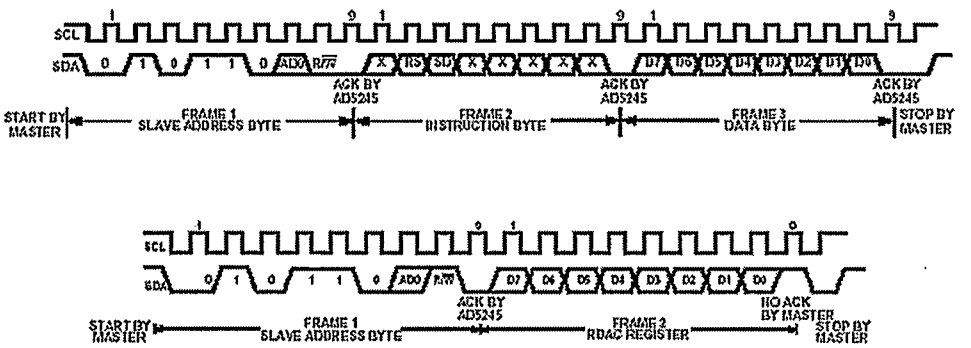


Figure 24. Digital Potentiometer I^2C Read/Write Protocol (Analog Devices, 2004a)

The bottom two waveforms in Figure 24 show a read operation. First, the slave address byte is sent, followed by an acknowledge bit and an R/W bit, which is high. Then

the data byte is received and the acknowledge bit is checked. For the digital potentiometer, the received data is its digital setting.

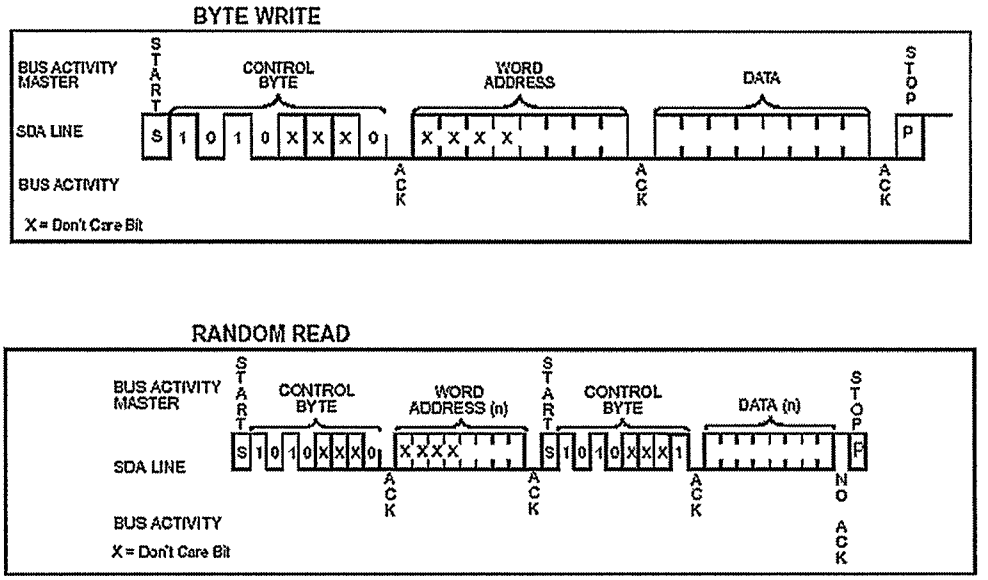
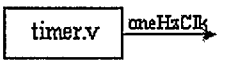


Figure 25. Serial EEPROM I^2C Read/Write Protocol (Microchip, 2003)

Figure 25 shows the I^2C accesses to the serial EEPROM, which follow a different protocol than that for the digital potentiometer. For write operations, the second byte is the address of the memory location to which the master will write data. For read operations, a write operation must take place to set the memory address, followed by a read operation to obtain the data from the memory. Again, any acknowledgement errors are flagged to sysControl.

3.3.1.6 One Hz Timer – Module timer.v

The timer is a simple clock divider, which divides the 32.768 kHz system clock to a precise 1 Hz clock, using 15 flip-flops.



3.3.1.7 Frequency Generator – Module freqGenerator.v

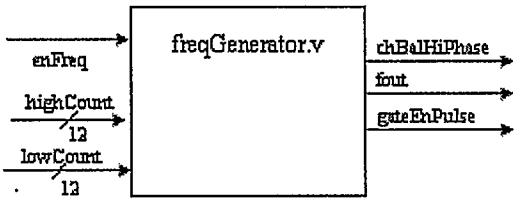


Figure 26. Block Diagram for freqGenerator.v

The frequency generator module contains a state machine with 6 states. This module generates the frequency signal to control the stimulation to the electrodes. The required frequency and duty cycle combinations required by Table 1 are created by the combination of the high count and low count inputs to the module. The high count specifies the duration, in system clock cycles, for which the stimulation signal will be active. Conversely the low count specifies the duration of how long the stimulation signal will be inactive. Figure 27 illustrates the synthesis of a single period, T, of the bi-polar stimulation signal, with the high and low count parameters.

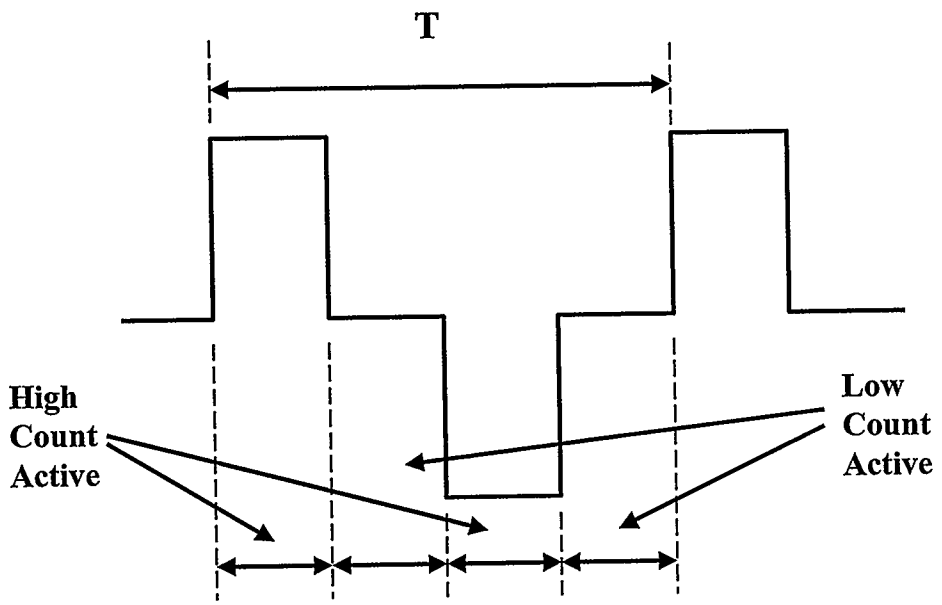


Figure 27. Stimulation Signal Synthesis with the High/Low Count Parameters

Since the high and low count inputs are 12 bits each, a wide range of frequencies and duty cycles can be programmed, far in excess of what is required in Table 1. Since the system clock frequency is quite low, it reasonable to question whether the frequency generator can meet the required frequencies in Table 1 accurately. A Matlab program was developed to optimize the high and low counts for any given frequency and duty cycle. This program is listed in Appendix E. To find the optimum value for the high and low count, the program calculates the percent errors in: the time the stimulation phase is high; the time the stimulation phase is low; and the frequency. The program selects the combination of high count and low count that minimizes the sum of the three absolute errors. Appendix D contains a table of optimum high and low count combinations for frequency and duty cycle combinations at values specified by Table 1. Table 6 shows the

example of the 50 Hz stimulation frequency, which is thought to be optimal in invoking gastrointestinal motility (Sanmiguel et al., 2004), at the full range of duty cycles.

Table 6. High and Low Count Values for 50 Hz and Varying Frequencies

Desired Frequency	Desired Duty Cycle	High Count	Low Count	% Error in freq	% Error in high count	% Error in low count
50	100	328	0	-0.1	-0.1	0.0
50	90	295	33	-0.1	0.0	-0.7
50	80	262	66	-0.1	0.1	-0.7
50	70	230	98	-0.1	-0.3	0.3
50	60	197	131	-0.1	-0.2	0.1
50	50	164	164	-0.1	-0.1	-0.1
50	40	131	197	-0.1	0.1	-0.2
50	30	98	230	-0.1	0.3	-0.3
50	20	66	262	-0.1	-0.7	0.1
50	10	33	295	-0.1	-0.7	0.0

The frequency generator module also has an output signal that indicates when the stimulation signal is charge-balanced so that stimulation can safely be turned off. This charge-balance is achieved simply through timing and not through any physical measurement of actual charge. Related to the charge-balance, this module also has an output signal that controls the polarity of bi-polar stimulation to the tissue.

3.3.1.8 Channel Switching Controller – chSwitcher.v

The channel switching controller module controls the signals to the analog switches, which ultimately control the electrical stimulation to the tissue. This module has two functions. The first function is to gate the signal to each stimulation channel based on the start and stop times for each channel. The second function is the control the polarity of the bi-polar stimulation signal.

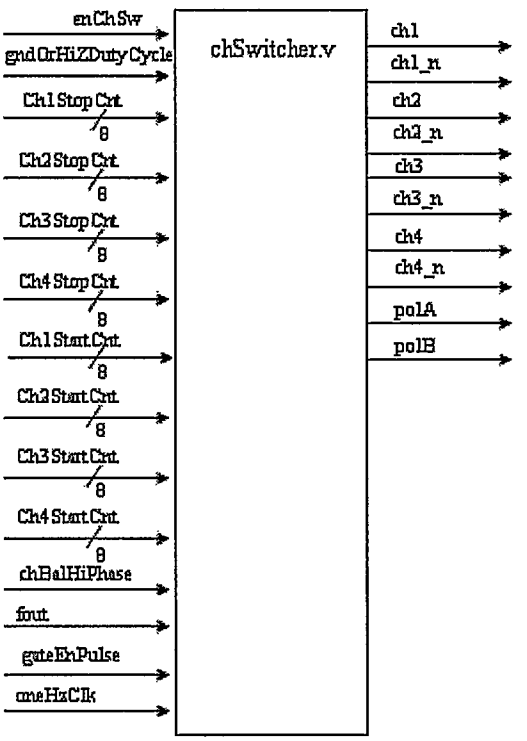


Figure 28. Block Diagram for chSwitcher.v

3.4 Analog Output Voltage Controller

Either the adjustable linear or switching voltage regulators can be used to deliver the voltage to stimulate the tissue. Using the feedback pin, the output of the voltage

regulators can be "programmed" to the correct output voltage by a suitable resistive network. A simplified, generic circuit diagram is shown in Figure 29.

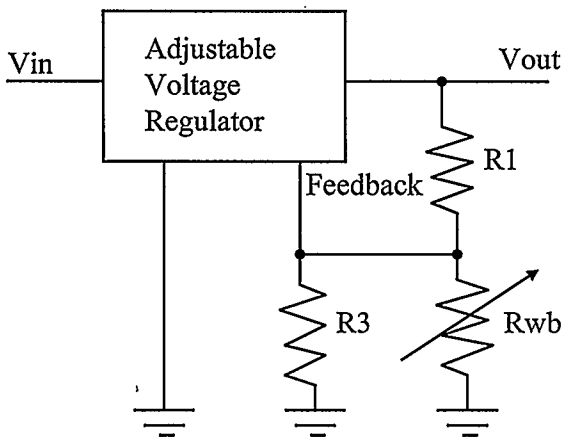


Figure 29. Circuit Diagram of Adjustable Voltage Regulator Programming

Mechanism

It should be noted that resistor R_{wb} is a digital potentiometer (Analog Devices, 2004b) which is programmed through the I^2C interface to adjust the output of the voltage regulator. The following equations describe the control mechanism of the adjustable voltage regulator.

$$V_{out} = V_{ref} \cdot (1 + \frac{R_1}{R_2}) \quad [4]$$

$$R_2 = \frac{R_3 \cdot R_{wb}}{R_3 + R_{wb}} \quad [5]$$

$$R_{wb}(D) = \frac{D}{256} \cdot R_{ab} + R_w \quad [6],$$

where D is the integer value programmed in the digital potentiometer

The circuit uses the following component values:

$R_1 = 44.6\ k\Omega$,

$R_3 = 1\ M\Omega$,

$R_{ab} = 100\ k\Omega$ (the full-scale resistance of the digital potentiometer),

$R_w = 50\ \Omega$,

$V_{ref} = 1.235\ V$.

Appendix F includes a table containing the voltage values for all 256 possible levels for the digital potentiometer. A subset of that table is in Table 7, which shows the digital potentiometer programming values meeting the output voltage requirements specified in Table 1.

Table 7. Digital Potentiometer Settings to Control Stimulation Voltage

D	R_{wb} (Ω)	$R_2 = R_3//R_{wb}$	V_{out} (V)
255	99659.375	90627.5	1.843
198	77393.75	71834.2	2.002
82	32081.25	31084.0	3.007
52	20362.5	19956.1	3.995
38	14893.75	14675.2	4.988
30	11768.75	11631.9	5.970
25	9815.625	9720.2	6.902
21	8253.125	8185.6	7.964
18	7081.25	7031.5	9.069
16	6300	6260.6	10.033

The voltage regulators are configured through two hardware jumpers to give three different ways of providing output voltages.

The first option is to allow just the linear regulator to provide the output voltage to the tissue. This method has the smallest component count and the output of the linear voltage regulator would be very smooth with good load transient response. This method also has the worst power dissipation.

The second option is to only use the switching regulator to provide the output voltage to the tissue. This method has the advantage of being power efficient at the expense of higher part count, increased output noise, and poorer load transient response.

The third option is to use the switching regulator as a pre-regulator to the linear regulator. This method would combine the best features of both prior solutions in terms of power efficiency, load transient response, and output noise. The drawbacks would be increased part count and diminished power efficiency, compared to the switching regulator only.

All three voltage regulator configuration options were tested and the results are presented in Chapter 4.

3.5 Serial EEPROM

The serial EEPROM Microchip, 2003) is accessed through the I^2C interface and holds the configuration data for the desired stimulation pattern. The serial EEPROM has a capacity of 128 bits, arranged as 16 x 8 bits.

3.6 Analog Switch

Three ADG452BR (Analog Devices, 2004b) analog switches are used to deliver the stimulation voltage to the tissue. As can be seen on the schematic diagram in Appendix B, one analog switch IC is used to switch the polarity of the stimulation to generate bi-polar stimulation. Switching the polarity of the stimulation is more power efficient than using a voltage inverter circuit, such as a charge pump. The other two analog switch ICs are used to deliver the voltage to the tissue. All the analog switches are controlled by the CPLD.

CHAPTER FOUR: TESTING RESULTS

This chapter describes the tests on implant power consumption with various typical load conditions.

To measure current and resistance, the Fluke 87 III multimeter (John Fluke Manufacturing Co., Everett, Washingtion) was used. To measure voltage, the multimeter was used only to verify the power supply voltage, and the Agilent 54621D Mixed Signal Oscilloscope (MSO) (Agilent Technologies, Colorado Springs, Colorado) was used to measure all other voltages. The power supply used to power the implant was the Hewlett Packard E3631A (Hewlett Packard, Palo Alto, California).

The power supply voltages to the implant emulated the DC voltage that would be received from the receiving coil's rectifier and filter circuit. In collaborative discussions on interface specifications between the implant and the belt, it was agreed that the target voltage to be delivered to the implant would be in the range of 10 to 20 V DC for implant current draws between 0 and 50 mA.

The implant is composed of two printed circuit boards (PCBs). The first PCB contains the CPLD, memory, and oscillator. Since this PCB includes all the digital circuitry, it is called the digital PCB. The second PCB contains the adjustable voltage regulators and analog switches to generate and deliver the electrical stimulation to the tissue. The second PCB is called the analog PCB.

4.1 Digital PCB Power Consumption

To measure the power consumption of the digital PCB, the digital PCB was electrically separated from the analog PCB in order to eliminate any loading effects from the analog PCB. The test configuration is shown in Figure 30.

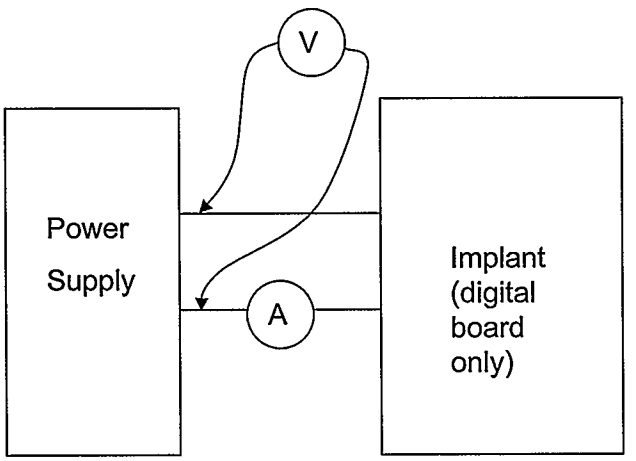


Figure 30. Test Set-Up to Measure Digital Circuit Power Consumption

Table 8 shows the power consumption of the digital circuit in the idle state, waiting for data transmission from the belt.

Table 8. Digital Circuit Power Consumption When Idle

Input Voltage (V)	Current (μA)	Power Consumption (mW)
10	168.3	1.683
20	169.5	3.39

Table 9 shows the power consumption of the digital circuit when it has passed the 120-second interval during which the system has waited for the power to stabilize and receive any data from the belt. After 120 seconds, the CPLD sends an error message every second to indicate that there is an I^2C error, because the digital potentiometers on the analog PCB have not asserted the acknowledge bit (because they are not connected to the digital PCB).

Table 9. Digital Circuit Power Consumption When Transmitting

Input Voltage (V)	Current (μA)	Power Consumption (mW)
10	192.2	1.922
20	192.6	3.85

It can be seen that the power consumption of the digital PCB compares closely to the CPLD power consumption prediction of 2.8 mW, made in Chapter 2.

4.2 Implant Power Consumption and Efficiency When Stimulating

The following power consumption and efficiency tests were performed with the digital and analog PCBs connected together. The colonic stimulation and worst-case

gastric stimulation cases were tested with the input voltage varying between 10 and 20 V. The test configuration is shown in Figure 31. For reference, the adjustable voltage regulator programming values are included in the tables below. For the switching regulator, the mnemonic is sRegVal, and for the linear regulator, the mnemonic is lRegVal. These mnemonics follow the convention in the Verilog code.

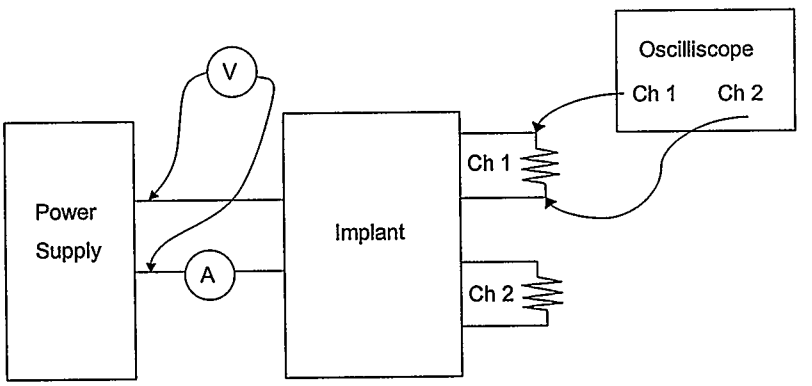


Figure 31. Test Set-Up to Measure Stimulation Power Efficiency

4.2.1 Colonic Testing Results

For the colonic tests, the stimulation parameters were taken from the research results of Sanmiguel et al (2004). Sanmiguel et al. cites a tissue impedance of 324 Ω ± 26 Ω while stimulating at a peak-to-peak rectangular voltage of between 8 to 10 volts, with either one or two channels active at any time. In these tests, a load of 326 Ω to each channel was used to emulate the tissue impedance.

In this testing, three phases of the implant are identified and tested:

1. when the implant is idle (not stimulating) and waiting for configuration data from the belt;
2. when only one channel is active;

3. when two channels are active.

The output voltage is varied between 4 and 5 volts peak and the input voltage is varied between 10 and 20 V in these tests. The tests include the different adjustable voltage regulator configurations as described in Chapter 3.

4.2.1.1 Colonic Example with No Stimulation

In the tables in this sub-section, there is no stimulation and the input voltage is varied between 10 and 20V, and the stimulation voltage is varied between 4 and 5 V.

Table 10 Colonic Test 1, No Stimulation

Input Voltage = 20V, Channel 1 Resistance = 326.3Ω, Channel 2 Resistance = 326.2Ω, Desired Stimulation Voltage = 4V, No Stimulation

Circuit	Measured Output Voltage (V)	sRegVal	lRegVal	Input Current (mA)	Power In (mW)	Power Out (mW)	Power Consumed by Implant (mW)	Efficiency (%)
Switching Reg Only	0	49	n/a	3.43	68.6	0.0	68.6	0.0
Switching &Linear Reg	0	37	45	3.22	64.4	0.0	64.4	0.0
Linear Reg Only	0	n/a	45	0.38	7.6	0.0	7.6	0.0

Table 11 Colonic Test 2, No Stimulation

Input Voltage = 20V, Channel 1 Resistance = 326.3Ω, Channel 2 Resistance = 326.2Ω, Desired Stimulation Voltage = 5V, No Stimulation

Circuit	Measured Output Voltage (V)	sRegVal	lRegVal	Input Current (mA)	Power In (mW)	Power Out (mW)	Calculated Power Consumed by Implant (mW)	Efficiency (%)
Switching Reg Only	0	34	n/a	3.44	68.8	0.0	68.8	0.0
Switching &Linear Reg	0	27	32	3.31	66.2	0.0	66.2	0.0
Linear Reg Only	0	n/a	32	0.41	8.2	0.0	8.2	0.0

Table 12 Colonic Test 3, No Stimulation

Input Voltage = 10V, Channel 1 Resistance = 326.3Ω, Channel 2 Resistance = 326.2Ω, Desired Stimulation Voltage = 4V, No Stimulation

Circuit	Measured Output Voltage (V)	sRegVal	lRegVal	Input Current (mA)	Power In (mW)	Power Out (mW)	Calculated Power Consumed by Implant (mW)	Efficiency (%)
Switching Reg Only	0	49	n/a	3.61	36.1	0.0	36.1	0.0
Switching &Linear Reg	0	42	45	3.27	32.7	0.0	32.7	0.0
Linear Reg Only	0	n/a	45	0.38	3.8	0.0	3.8	0.0

Table 13 Colonic Test 4, No Stimulation

Input Voltage = 10V, Channel 1 Resistance = 326.3Ω, Channel 2 Resistance = 326.2Ω, Desired Stimulation Voltage = 5V, No Stimulation

Circuit	Measured Output Voltage (V)	sRegVal	lRegVal	Input Current (mA)	Power In (mW)	Power Out (mW)	Calculated Power Consumed by Implant (mW)	Efficiency (%)
Switching Reg Only	0	34	n/a	3.41	34.1	0.0	34.1	0.0
Switching &Linear Reg	0	27	32	3.38	33.8	0.0	33.8	0.0
Linear Reg Only	0	n/a	32	0.4	4.0	0.0	4.0	0.0

It is observed from the preceding tables that the switching regulator adds approximately 3 mA of current consumption when the implant is not stimulating.

4.2.1.2 Colonic Example with Channel 1 On

In the tables in this sub-section, one channel is on and the input voltage is varied between 10 and 20V, and the stimulation voltage is varied between 4 and 5 V.

Table 14 Colonic Test 1, Channel 1 On

Input Voltage = 20V, Channel 1 Resistance = 326.3Ω, Channel 2 Resistance = 326.2Ω, Desired Stimulation Voltage = 4V, Only Channel 1 On

Circuit	Measured Output Voltage (V)	sRegVal	lRegVal	Input Current (mA)	Power In (mW)	Power Out (mW)	Power Consumed by Implant (mW)	Efficiency (%)
Switching Reg Only	3.969	49	n/a	8.98	179.6	48.3	131.3	26.9
Switching &Linear Reg	4	37	45	10.16	203.2	49.0	154.2	24.1
Linear Reg Only	4	n/a	45	12.76	255.2	49.0	206.2	19.2

Table 15 Colonic Test 2, Channel 1 On

Input Voltage = 20V, Channel 1 Resistance = 326.3Ω, Channel 2 Resistance = 326.2Ω, Desired Stimulation Voltage = 5V, Only Channel 1 On

Circuit	Measured Output Voltage (V)	sRegVal	lRegVal	Input Current (mA)	Power In (mW)	Power Out (mW)	Calculated Power Consumed by Implant (mW)	Efficiency (%)
Switching Reg Only	5	34	n/a	13.3	266.0	76.6	189.4	28.8
Switching &Linear Reg	5	27	32	13.17	263.4	76.6	186.8	29.1
Linear Reg Only	4.969	n/a	32	15.81	316.2	75.7	240.5	23.9

Table 16 Colonic Test 3, Channel 1 On

Input Voltage = 10V, Channel 1 Resistance = 326.3Ω, Channel 2 Resistance = 326.2Ω, Desired Stimulation Voltage = 4V, Channel 1 On

Circuit	Measured Output Voltage (V)	sRegVal	lRegVal	Input Current (mA)	Power In (mW)	Power Out (mW)	Calculated Power Consumed by Implant (mW)	Efficiency (%)
Switching Reg Only	3.906	49	n/a	10.9	109.0	46.8	62.2	42.9
Switching &Linear Reg	3.969	42	45	11.61	116.1	48.3	67.8	41.6
Linear Reg Only	4	n/a	45	12.65	126.5	49.0	77.5	38.8

Table 17 Colonic Test 4, Channel 1 On

Input Voltage = 10V, Channel 1 Resistance = 326.3Ω, Channel 2 Resistance = 326.2Ω, Desired Stimulation Voltage = 5V, Channel 1 On

Circuit	Measured Output Voltage (V)	sRegVal	lRegVal	Input Current (mA)	Power In (mW)	Power Out (mW)	Calculated Power Consumed by Implant (mW)	Efficiency (%)
Switching Reg Only	4.906	34	n/a	13.55	135.5	73.8	61.7	54.4
Switching &Linear Reg	4.969	27	32	14.47	144.7	75.7	69.0	52.3
Linear Reg Only	4.906	n/a	32	15.68	156.8	73.8	83.0	47.0

The preceeding tables clearly illustrate that the various regulator options are very close in terms of efficiency, but the switching regulator options have a slight advantage over the linear regulator option. For all regulator options, the highest efficiency is when the difference between the input and output voltages are smaller rather larger.

4.2.1.3 Colonic Example with Two Channels On

In the tables in this sub-section, there are two channels on and the input voltage is varied between 10 and 20V, and the stimulation voltage is varied between 4 and 5 V.

Table 18 Colonic Test 1, Channel 1 and 2 On

Input Voltage = 20V, Channel 1 Resistance = 326.3Ω, Channel 2 Resistance = 326.2Ω, Desired Stimulation Voltage = 4V, Channel 1 and 2 On

Circuit	Measured Output Voltage (V)	sRegVal	lRegVal	Input Current (mA)	Power In (mW)	Power Out (mW)	Calculated Power Consumed by Implant (mW)	Efficiency (%)
Switching Reg Only	3.906	49	n/a	11.97	239.4	93.5	145.9	39.1
Switching &Linear Reg	3.969	37	45	12.84	256.8	96.6	160.2	37.6
Linear Reg Only	3.938	n/a	45	24.93	498.6	95.1	403.5	19.1

Table 19 Colonic Test 2, Channel 1 and 2 On

Input Voltage = 20V, Channel 1 Resistance = 326.3Ω, Channel 2 Resistance = 326.2Ω, Desired Stimulation Voltage = 5V, Channel 1 and 2 On

Circuit	Measured Output Voltage (V)	sRegVal	lRegVal	Input Current (mA)	Power In (mW)	Power Out (mW)	Calculated Power Consumed by Implant (mW)	Efficiency (%)
Switching Reg Only	4.969	34	n/a	15.08	301.6	151.4	150.2	50.2
Switching &Linear Reg	4.938	27	32	16.23	324.6	149.5	175.1	46.1
Linear Reg Only	4.875	n/a	32	30.99	619.8	145.7	474.1	23.5

Table 20 Colonic Test 3, Channel 1 and 2 On

Input Voltage = 10V, Channel 1 Resistance = 326.3Ω, Channel 2 Resistance = 326.2Ω, Desired Stimulation Voltage = 4V, Channel 1 and 2 On

Circuit	Measured Output Voltage (V)	sRegVal	lRegVal	Input Current (mA)	Power In (mW)	Power Out (mW)	Calculated Power Consumed by Implant (mW)	Efficiency (%)
Switching Reg Only	3.844	49	n/a	15.54	155.4	90.6	64.8	58.3
Switching &Linear Reg	3.906	42	45	17.39	173.9	93.5	80.4	53.8
Linear Reg Only	3.906	n/a	45	24.62	246.2	93.5	152.7	38.0

Table 21 Colonic Test 4, Channel 1 and 2 On

Input Voltage = 10V, Channel 1 Resistance = 326.3Ω, Channel 2 Resistance = 326.2Ω, Desired Stimulation Voltage = 5V, Channel 1 and 2 On

Circuit	Measured Output Voltage (V)	sRegVal	lRegVal	Input Current (mA)	Power In (mW)	Power Out (mW)	Calculated Power Consumed by Implant (mW)	Efficiency (%)
Switching Reg Only	4.844	34	n/a	21.01	210.1	143.8	66.3	68.5
Switching &Linear Reg	4.875	27	32	23.25	232.5	145.7	86.8	62.7
Linear Reg Only	4.813	n/a	32	30.62	306.2	142.0	164.2	46.4

It can be seen from the preceding tables that the switching regulator options are showing significantly increased efficiency compared to the linear regulator only. This is the result of the increased efficiency of the switching regulator at higher load currents. Again, for all the regulator options, the highest efficiency is when the difference between the input and output voltages are smaller rather larger.

4.2.2 Worst-Case Gastric Test Results

As discussed in Chapter 2, gastric SNGES may have higher power requirements than colonic SNGES. The worst-case test for the gastric SNGES was to apply a load resistance of 100 Ω to a single channel with an output stimulation voltage of 10V (20 V peak-to-peak). In these tests, a 101.5 Ω resistance was used for the load on channel 1. The input voltage was varied between 13 and 20 V. The reason the lower input voltage

was increased from 10 V in the previous colonic SNGES tests to 13 V was because the dropout voltage of the switching regulator increased as the load current increased. This means that, as the load current increases, the minimum required difference between the input and output voltage of the voltage regulator increases. The same behaviour was observed for the linear regulator, but not to the same degree as the switching regulator.

4.2.2.1 Worst-Case Gastric Example with No Stimulation

In the tables in this sub-section, there is no stimulation and the input voltage is varied between 10 and 20V, and the stimulation voltage is 10 V.

Table 22 Gastric Test 1, No Stimulation

Input Voltage = 20V, Channel 1 Resistance = 101.5Ω, Channel 2 Resistance = n/a, Desired Stimulation Voltage = 10V, No Stimulation

Circuit	Measured Output Voltage (V)	sRegVal	lRegVal	Input Current (mA)	Power In (mW)	Power Out (mW)	Calculated Power Consumed by Implant (mW)	Efficiency (%)
Switching Reg Only	0	11	n/a	3.62	72.4	0.0	72.4	0.0
Switching &Linear Reg	0	10	10	3.66	73.2	0.0	73.2	0.0
Linear Reg Only	0	n/a	10	0.4	8.0	0.0	8.0	0.0

Table 23 Gastric Test 2, No Stimulation

Input Voltage = 13V, Channel 1 Resistance = 101.5Ω, Channel 2 Resistance = n/a, Desired Stimulation Voltage = 10V, No Stimulation

Circuit	Measured Output Voltage (V)	sRegVal	lRegVal	Input Current (mA)	Power In (mW)	Power Out (mW)	Calculated Power Consumed by Implant (mW)	Efficiency (%)
Switching Reg Only	0	11	n/a	3.93	51.1	0.0	51.1	0.0
Switching &Linear Reg	0	10	10	4.02	52.3	0.0	52.3	0.0
Linear Reg Only	0	n/a	10	0.53	6.9	0.0	6.9	0.0

It is observed that power consumption in the above tables is very similar to that shown for colonic SNGES in the previous section, with no stimulation. It can also be noted that even though there is no voltage applied to the load, programming the voltage regulator to a higher output voltage increases its power consumption.

4.2.2.2 Worst-Case Gastric Example with One Channel On

In the tables in this sub-section, one channel is stimulating, the input voltage is varied between 10 and 20V, and the stimulation voltage is 10 V.

Table 24 Gastric Test 1, Channel 1 On

Input Voltage = 20V, Channel 1 Resistance = 101.5Ω, Channel 2 Resistance = n/a, Desired Stimulation Voltage = 10V, Channel 1 On

Circuit	Measured Output Voltage (V)	sRegVal	lRegVal	Input Current (mA)	Power In (mW)	Power Out (mW)	Calculated Power Consumed by Implant (mW)	Efficiency (%)
Switching Reg Only	9.938	11	n/a	63	1260.0	973.0	287.0	77.2
Switching &Linear Reg	10.125	10	10	71.4	1428.0	1010.0	418.0	70.7
Linear Reg Only	10.063	n/a	10	106	2120.0	997.7	1122.3	47.1

Table 25 Gastric Test 2, Channel 1 On

Input Voltage = 13V, Channel 1 Resistance = 101.5Ω, Channel 2 Resistance = n/a, Desired Stimulation Voltage = 10V, Channel 1 On

Circuit	Measured Output Voltage (V)	sRegVal	lRegVal	Input Current (mA)	Power In (mW)	Power Out (mW)	Calculated Power Consumed by Implant (mW)	Efficiency (%)
Switching Reg Only	9.831	11	n/a	91.7	1192.1	952.2	239.9	79.9
Switching &Linear Reg	10.063	10	10	103.2	1341.6	997.7	343.9	74.4
Linear Reg Only	9.938	n/a	10	103.8	1349.4	973.0	376.4	72.1

As can be noted from the above tables, the switching regulator options are superior in efficiency compared to the linear regulator option. Since the load current is dramatically increased for the worst-case gastric SNGES case, the switching regulator operates with high efficiency. When the input voltage is close to the output voltage, the linear regulator performs almost as well as the switching regulator options.

4.2.3 Discussion of Test Results

The testing showed that the "switching regulator only" option was the most power efficient, especially when there was a large difference in the input to output voltages. On the oscilloscope, greater voltage ripple and a switching transient were observed in the switching regulator, compared to the linear regulator. Since the ripple and transient

effects may lead to a charge imbalance in the tissue, using the switching regulator as a pre-regulator to the linear regulator is recommended. This option was recommended over the linear regulator because it minimizes the implant power consumption which is important in minimizing heat to the surrounding tissue.

Since the efficiency of all the regulator options is strongly dependent on the difference between the input and output voltages, the control of the input voltage becomes an important design feature. At the moment, the transcutaneous energy link is designed with open-loop control. This means that the belt attempts to transfer as much energy as is required to the implant to cover the worst-case requirements. This results in input voltages that can be significantly higher than what is required for stimulation. A compelling argument can be made for the closed-loop control of the transcutaneous energy transfer. This means that a feedback signal must be sent to the belt to indicate the input voltage at the implant. The belt could then adjust the transcutaneous energy transfer such that an optimal input voltage could be maintained at the implant. Using this method, the switching regulator could be eliminated and the linear regulator alone could be used. Using closed-loop control, it is conceivable that the linear regulator alone could be more efficient than using a switching regulator.

CHAPTER FIVE: CONCLUSIONS

My neurostimulator design research was motivated to address a major deficiency in a newly proposed and very promising stimulation method for imposing a pre-designed and electrically imposed motility pattern in the gastrointestinal tract. My work was aimed at designing a transcutaneously-powered implantable device to facilitate this stimulation method. The absence of such an implantable device was a critical impediment to the testing of the stimulation method on chronic experimental animals and subsequently on humans.

The outcome of this project is a successful collaboration in the design and implementation of an innovative transcutaneously-powered device that is small and safe enough to be implanted in an animal. The device conforms to the stimulation algorithm used in the SNGES method to restore impaired gastrointestinal motility.

The charge balance problem was addressed in the system controller design. Testing the charge balance was outside of the scope of this research. However, design decisions were made to optimize the charge balance. The controller was designed to deliver precisely timed and identical positive and negative phases of the stimulation pattern to the gastrointestinal tissue. The continuation of this research will focus on determining whether the design successfully addresses the problem of sufficient charge balance through chronic animal testing.

Design decisions were also made to minimize the power dissipation of the neurostimulator and these will need to be tested in future research. It is anticipated that the development of a closed-loop transcutaneous energy transfer system will significantly improve the power efficiency of the implant.

A detailed design procedure was demonstrated in the design of this device which addresses key issues such as charge balance, tissue heating, and biocompatibility. This design procedure lays the foundation for the future architecture of a mixed-signal ASIC based neurostimulator that would lead to further minimization of power consumption and size. The Verilog code used in this design can easily be ported to the mixed-signal ASIC which will speed the development of the ASIC design, thus demonstrating, the functionality of the code.

An analysis of the system clock frequency and the SNGES requirements led to the reduction and optimization of the system clock frequency. The choice of a CPLD to prototype the system controller design led to further power reduction, compared to using standard microcontrollers. The frequency and system controller decisions led to comparatively small optimizations in power reduction. The larger optimizations in power reduction came from implementing an adjustable switching regulator to directly deliver electrical stimulation to the tissue.

The neurostimulator was implemented as a prototype and various circuit configurations were tested for power efficiency. The test results were compared and the switching regulator acting as a pre-regulator for the linear regulator was found to be optimal in terms of power efficiency and smooth output voltage. The results of these tests will form a benchmark for future research in the design of neurostimulators that implement the SNGES method.

Additional work is required to complete the integration with the transcutaneous energy transfer and communications link circuitry. Then, the implant will be conformally coated in silicone and tested with the complete transcutaneous energy transfer and

communications link circuitry. Animal implantation and testing will follow the successful laboratory testing of the packaged implant.

BIBLIOGRAPHY

Abell, T., McCallum, R., Hocking, M., Koch, K., Abrahamsson, H., LeBlanc, I., Lindberg, G., Konturek, J., Nowak, T., Quigley, E. M. M., Tougas, G., & Starkebaum, W. (2003). Gastric electrical stimulation for medically refractory gastroparesis. *Gastroenterology*, 125, 421-428.

Akwari, O.E., Kelly, K.A., Steinbach, J.H., & Code, C.F. (1975). Electric pacing of intact transected canine small intestine and its computer model. *Am J Physiol,* 229, 1188-1197.

Amaris M., Rashev P.Z., Mintchev M.P. & Bowes K.L. (2002) Microprocessor-controlled movement of solid colonic content using sequential neural electrical stimulation. *Gut,* 50, 475-479.

Analog Devices Incorporated (2004a). 256-Position I^2C compatible digital potentiometer. retrieved Dec 18, 2004, from http://rocky.digikey.com/WebLib/Analog%20Devices/Web%20Data/AD5245.pdf

Analog Devices Incorporated (2004b).ADR451/ADG452/ADG453: LC^2MOS 5 Ω R_{on} SPST switches. retrieved Dec 18, 2004, from http://www.analog.com/UploadedFiles/Data_Sheets/5131184922974ADG451_2_3_b.pdf

Atmel Corporation, (2004). 8-bit AVR microcontroller with 2kbytes in-system programmable flash. retrieved Dec. 27, 2004, from http://www.atmel.com/dyn/resources/prod_documents/doc2543.pdf

Bellahsene, B. E., Lind, C. D., Schirmer, B. D., Updike, O.L., McCallum, R. W. (1992). Acceleration of gastric emptying with electrical stimulation in a canine model of gastroparesis. *American Journal of Physiology*, 252, G826-G834

Bilgutay, A. M., Wingrove R., Grifen W. O., Bonnabeau, R. C., & Lillehei, C.W. (1963). Gastro-intestinal pacing: a new concept in the treatment of ileus. *Annals of Surgery*, 158(3), 139-144.

Binnie, N.R., Smith, A.N., Creasey, G.H., & Edmond, P. (1991) Constipation associated with chronic spinal cord injury: The effect of pelvic prasympathetic stimulation by the Brindley stimulator. *Paraplegia,* 29, 463-469.

Bortolotti, M. (2002). The "electrical way" to cure gastroparesis. *The American Journal of Gastroenterolgy*, 97(8), 1874-1883.

Brindley, G.S., Polkey, C.E., & Rushton, D.N. (1998). The Finetech-Brindley bladder controller: notes for surgeons and physicians. London: Maudsley Hospital.

Bruninga, K., Riedy, L., Keshavarzian, A., & Walter, J. (1998). The effect of electrical stimulation on colonic transit following spinal cord injury in cats. *Spinal Cord*, 36, 847-53.

Castagnoli, G., Braeuer, S., and Herrman M. (1993). Optimization of cyclic redundancy-check codes with 24 and 32 parity bits. *IEEE Transactions on Communications*, 41(6), 883-892.

Cranley B., Kelly K.A., Go V.L.W., & McNichols L.A. (1983). Enhancing the anti-dumping effect of Roux gastrojejunostomy with intestinal pacing. *Annals of Surgery,* 198, 516-24.

Devinder, K., & Gustavsson, S. (1988). *An illustrated guide to gastrointestinal motility.* New York: Wiley.

Digestive disease statistics. (1999). retrieved Dec. 19, 2004, from National Digestive Diseases Information Clearinghouse Web site: http://digestive.niddk.nih.gov/statistics/statistics.htm

Eagon, J. C., Kelly, K. A. (1995). Effect of electrical stimulation on gastric electrical activity, motility and emptying. *Neurogastroenterology & Motility*. 7(1), 39-45.

Folwaczny, C., Riepl, R., Tschop, M. & Landgraf, R. (1999). Gastrointestinal involvement in patients with diabetes mellitus: Part 1. Epidemiology, pathophysiology, clinical findings. *Zeitschrift fur Gastroenterologie*, 37(9), 803-15.

Grundfest-Broniakowski, S., Moritz, A., Ilyes, L., Jacobs, G., Kasick, J., et al. (1988a). Voluntary control of an ileal pouch by coordinated electrical stimulation: a pilot study in the dog. *Diseases of the Colon and Rectum,* 31, 261-7.

Grundfest-Broniakowski, S., Moritz, A., Olsen, E., Kasick, J., et al.(1988b). Electrical control of intestinal reservoirs in a chronic dog model. *ASAIO Transactions,* 34, 664-8.

Hanani, M., & Freund, H. R. (2000). Interstitial cells of Cajal - their role in pacing and signal transmission in the digestive system. *Acta Physiologica Scandinavica*, 170(3), 177-190.

Hasler, W. L. (2000). The brute force approach to electrical stimulation of gastric emptying: a future treatment for refractory gastroparesis. *Gastroenterology*, 118, 433-442.

Hocking, M. P., Vogel, S. B., Sninsky, C. A. (1992). Human gastric myoelectric activity and gastric emptying following gastric surgery and with pacing. *Gastroenterolgy*, 103(6) 1811-1816.

Hollerbach, S., Tougas, G., Frieling, T., Enck, P., et al. (1997). Cerebral evoked responses to gastrointestinal stimulation. *Critical Reviews in Biomedical Engineering*, 25, 203-42.

Hrdlicka, G. (2002). Implantable neurostimulators and the EMC environment. *2002 IEEE International Symposium on Electromagnetic Compatibility*, 2(August), 916-919.

Hughes, S.F., Scott, S.M., Pilot, M.A., & Williams, N.S. (1995). Electrically stimulated colonic reservoir for total anorectal reconstruction. *British Journal of Surgery,* 82, 1321-1326.

Huizinga, J.D. (1986). Electrophysiology of human colon motility in health and disease. *Clinics in Gastroenterology*, 15(4), 879-901.

Johnson, L. R. (1994). *Physiology of the gastrointestinal tract*. New York: Raven Press.

Jones, K. L., Russo, A., Berry, M. K., Stevens, J. E., Wishart, & J. M., Horowitz, M. (2002). A longitudinal study of gastric emptying and upper gastrointestinal symptoms in patients with diabetes mellitus. *The American Journal of Medicine*, 113(6), 449-455.

Keating, M. and Bricaud, P. (1999). *Reuse methodology manual for system-on-chip designs*. Boston, Massachusetts: Kluwer Academic Publishers.

Kong, M., Horowitz, M., Jones, K. L., Wishart, J. M., & Harding, P. E. (1999). Natural history of diabetic gastroparesis. *Diabetes Care*, 22(3), 503-507.

Koopman, P. (2002). 32-bit cyclic redundancy codes for internet applications. *Proceedings. International Conference on Dependable Systems and Networks*, 459-468.

Koopman, P., Chakravarty, T. (2004). Cyclic redundancy code (CRC) polynomial selection for embedded networks. *Proceedings. International Conference on Dependable Systems and Networks*, 145-154.

Lin, Y., Sanmiguel, C. P., Turner, L. E., Soffer, E., & Mintchev, M. P. (2003). Hardware-software co-design of portable functional gastrointestinal stimulator system. *Journal of Medical Engineering and Technology*, 27(4), 164-177.

Lin, Z., Forester, J., Sarosiek, I., & McCallum, R. W. (2003). Treatment of gastroparesis with electrical stimulation. *Digestive Diseases and Sciences*, 48(5), 837-848.

Lin, Z., Forester, J., Sarosiek, I., & McCallum, R. W. (2004). Effect of high-frequency gastric electrical stimulation on gastric myoelectric activity in gastroparetic patients. *Neurogastroenterology and Motility*, 16, 205-212.

Livingston, E.H. & Passaro E.P., Jr. (1990). Postoperative Ileus. *Digestive Diseases and Sciences*, 5(1), 121-32.

MacDonagh, R.P., Sun, W.M., Smallwood, R., Forster, D., & Read, N.W. (1990). Control of defecation in patients with spinal injuries by stimulation of sacral anterior nerve roots. *British Medical Journal,* 300, 1494-1497.

Marieb, E. (1998). *Human anatomy and physiology*. Menlo Park, CA: Benjamin/Cummings.

Mayer, W., Bijak, M., Rafolt, D., Sauermann, S., Unger, E., & Lanmuller, H. (2001). Basic design and construction of the Vienna FES implants: existing solutions and prospects for new generations of implants. *Medical Engineering & Physics*, 23, 53-60.

McCallum, R. W., & George, S. J. (2001). Gastroparesis. Clinical Perspectives in *Gastroenterolgy*, May/June, 147-154.

Medtronic Inc., (n.d.). Enterra therapy for gastroparesis. retrieved Nov. 22, 2004, from http://www.medtronic.com/neuro/gastro/enterra/Enterra_MD%20Brochure_US.pdf.

Medtronic Inc., (n.d.). Enterra therapy. retrieved Nov. 19, 2004, from http://www.medtronic.com/neuro/gastro/enterra/enterra.html

Microchip Technology Incorporated (2003). 24AA00/24LC00/24C00 128 bit I^2C bus serial EEPROM. retrieved Dec 18, 2004, from http://rocky.digikey.com/WebLib/Microchip/Web%20Data/24AA00,24LC00,24C00.pdf

Mintchev, M. P. (n. d.). Stomach. *Electrogastrography, Methodolgy and Clinical Applicability, WWW Cyberzine*. Retrieved January 10, 2005, from http://www.enel.ucalgary.ca/People/Mintchev/stomach.htm.

Mintchev, M. & Bowes, K. (1997). Computer model of gastric electrical stimulation. *Annals of Biomedical Engineering*, 25, 726-730.

Mintchev, M. P., Sanmiguel, C. P., Amaris, M, & Bowes, K. L. (2000). Microprocessor controlled movement of solid gastric content using sequential neural electrical stimulation. *Gastroenterology*, 118, 258-263.

Mintchev, M. P., Sanmiguel, C. P., Otto, S. G., & Bowes, K. L. (1998). Microprocessor controlled movement of liquid gastric content using sequential neural electrical stimulation. *Gut*, 43, 607-611.

National Semiconductor, (2003). LM2671 simple switcher power converter high efficiency 500mA step-down voltage regulator with features. retrieved Dec 17, 2004, from http://www.national.com/ds/LM/LM2671.pdf

O'Brien, M.D. & Phillips, S.F. (1996). Colonic motility in health and disease. *Gastroenterology Clinics of North America*, 25(1), 147-62.

Philips Electronics Corporation (2004). Protocols I^2C. retrieved Dec 18, 2004, from http://www.semiconductors.philips.com/markets/mms/protocols/i2c/

Phillips, S. F. (Ed.). (1991). *The large intestine: physiology, pathophysiology, and surgery*. New York: Raven Press.

Plonsey, R., & Barr, R. (1988). *Bioelectricity: a quantitative approach*. New York: Plenum Press.

Rashev P.Z., Amaris M., Bowes K.L. & Mintchev M.P. (2002a). Microprocessor-controlled Colonic Peristalsis:Dynamic Parametric Modeling in Dogs. *Digestive Diseases and Sciences,* 47(5), 1034-48.

Rashev, P. Z., Bowes, L. B., & Mintchev, M. P. (2002b). Three-dimensional object-oriented modeling of the stomach for the purpose of microprocessor-controlled

functional stimulation. *IEEE Transactions on Information Technology in Biomedicine*, 6(4), 296-309.

Rashev, P. Z., Mintchev, M. P., & Bowes, L. B. (2001). Three-dimensional static parametric modelling of phasic colonic contractions for the purpose of microprocessor-controlled functional stimulation. *Journal of Medical Engineering & Technology*, 25(3), 85-96.

Ramabadran, T. V., Gaitonde, S. S., (1988). A tutorial on CRC computations. *IEEE Micro*.8(4), 62-74.

Ratani, R.S., Yazaki, E., Scott, M., Pilot, M.A., & Williams, N.S. (1997). Electrically stimulated smooth muscle neosphincter. *British Journal of Surgery,* 84, 1286-1289.

Sanmiguel, C. P., Casillas, S., Senagore, A., Mintchev, M. P., & Soffer, E. (2004). Neural gastrointestinal electrical stimulation enhances colonic motility in a chronic canine model of constipation. Manuscript, submitted to *Gut*.

Sarr, M.G., Kelly, K.A., & Gladen, H.E. (1981). Electrical control of canine jejunal propulsion. *American Journal of Physiology. (Gastrointesinal. Liver Physiology.),* 240, G355-G360.

Smith, D. J. (1996). *HDL chip design: a practical guide for designing, synthesizing and simulating ASICs and FPGAs using VHDL or Verilog*. Madison, Alaska: Doone Publications.

Uzunoglu, N. K. & Nikita, K. S. (1988). Estimation of temperature distribution inside tissues heated by interstitial RF electrode hyperthermia systems. *IEEE Transactions on Biomedical Engineering*, 35(4), 250-256.

Varma, J.S., Binnie, N., Smith, A.N., Creasey, G.H., & Edmond, P. (1986). Different effects of sacral anterior root stimulation on anal sphincter and colorectal motility in spinally injured man. *British Journal of Surgery*, 73, 478-482.

Wakerly, J. F. (1994). *Digital design: principles and practices. 2nd ed.* Englewood Cliffs, NJ: Prentice-Hall.

Williams, P. (Ed.). (1995). *Gray's anatomy: the anatomical basis of medicine and surgery*. Edinburgh: Churchill Livingstone.

Wise, K. D., Anderson, D. J., Hetke, J. F., Kipke, D. R., & Najafi, K. (2004). Wireless implantable Microsystems: high-density electronic interfaces to the nervous system. Proceedings of the IEEE, Special Issue on: *BIOMEDICAL APPLICATIONS FOR MEMS AND MICROFLUIDICS*, 92(1), 76-96.

Xilinx Incorporated, (2002). Understanding the CoolRunner-II Logic Engine. retrieved Dec 17, 2004, from http://direct.xilinx.com/bvdocs/appnotes/xapp376.pdf

Xilinx Incorporated, (2003). Powering CoolRunner-II CPLDs. retrieved Dec 17, 2004, from http://direct.xilinx.com/bvdocs/appnotes/xapp389.pdf

Xilinx Incorporated, (2004a). CoolRunner-II CPLD Family. retrieved Dec 17, 2004, from http://direct.xilinx.com/bvdocs/publications/ds090.pdf

Xilinx Incorporated, (2004b). XC2C512 CoolRunner-II CPLD. retrieved Dec 17, 2004, from http://direct.xilinx.com/bvdocs/publications/ds096.pdf

APPENDIX A: VERILOG CODE

A.1. Module paramFile.v

```
// Parameters for gastroCPLD design

/*
Note:
Because this "paramFile.v" file is included with all/almost all of the RTL
files in this design, repeated "Macro 'timerBits' redefined" etc, warnings
are produced when compiling in Xilinx. Because the same "paramFile.v" file
is used for all the RTL files, this warning can be safely ignored.
*/



// RTL compilation control
//  `define simulate  // uncomment this line for simulation,
                      // comment for correct RTL compilation (for the oneHzClk signal)

// Bitwidths in modules

// timer.v
   `define timerBits 15

// freqGenerator.v
   `define pwmCountBits 12

// chSwitcher.v
   `define chSwTimerBits 8

// register.v
   `define registerBits 144 // 105 for internal settings, 32 for CRC, 7 spare
   `define numRegisterBytes 144/8 // 105 for internal settings, 32 for CRC, 7 spare


//sysControl
   `define secondsCountBits 7 // 6 = 128 seconds count

   // messages
   // note, nibble commands are coded in Manchester format: 0 = 01, 1 = 10
```

A.2. Module top.v

```
`timescale 1 ns/1 ps

module top(
           clk
           ,SDA,SCL
           ,rxD,txD
           ,sRegOn,lRegOn_n,lRegErr_n
           ,ch1,ch2,ch3,ch4
           ,ch1_n,ch2_n,ch3_n,ch4_n
           ,polA,polB
           );

  `include "paramFile.v"
  /*
  Because this "paramFile.v" file is included with all/almost all of the RTL
  files in this design, repeated "Macro 'timerBits' redefined" etc, warnings
  are produced when compiling in Xilinx. Because the same "paramFile.v" file
  is used for all the RTL files, this warning can be safely ignored.
  */

  input clk;

  inout  SDA;
  output SCL;

  input  rxD;
  output txD;

  output sRegOn;
  output lRegOn_n;
  input  lRegErr_n;

  output ch1;
  output ch2;
  output ch3;
  output ch4;
  output ch1_n;
  output ch2_n;
  output ch3_n;
  output ch4_n;
  output polA;
  output polB;

  wire clk;
  wire rst_n;
  wire enFreq;
  wire [`pwmCountBits-1:0] highCount;
  wire [`pwmCountBits-1:0] lowCount;
  wire fout;
  wire oneHzClk;
  wire enChSw;
  wire [`chSwTimerBits-1:0] ch1StartCnt ;
  wire [`chSwTimerBits-1:0] ch1StopCnt  ;
  wire [`chSwTimerBits-1:0] ch2StartCnt ;
  wire [`chSwTimerBits-1:0] ch2StopCnt  ;
  wire [`chSwTimerBits-1:0] ch3StartCnt ;
  wire [`chSwTimerBits-1:0] ch3StopCnt  ;
  wire [`chSwTimerBits-1:0] ch4StartCnt ;
  wire [`chSwTimerBits-1:0] ch4StopCnt  ;
  wire chBalHiPhase;
  wire ch1;
  wire ch2;
  wire ch3;
  wire ch4;
  wire ch1_n;
  wire ch2_n;
```

```
  wire ch3_n;
  wire ch4_n;
  wire polA;
  wire polB;

  wire gndOrHiZDutyCycle;
  wire gateEnPulse;

 // wire serialData;
  wire [15:0] i2cAddr;
  wire i2cComplete;
  wire i2cRd;
  wire i2cWr;
  wire i2cEn;

  wire rd;
  wire wr;
  wire [7:0] uartDout;
  wire dataReady;
  wire [7:0] uartDin;
  wire uartEn;

//  wire crcSerialShift;
  wire crcPass;

  wire serialShift;

  wire enCrc;

  wire uartErr;

  wire i2cDataOut;
  wire regDataOut;
  wire regLoad;
  wire i2cErr;
  wire txComplete;
  wire chSwComplete;
  wire ACKclk;
  wire [4:0] currentState,nextState;
  wire manErr;
  wire demodErr;

// Reset Circuit
  resetBlock resetBlock(
    .rst_n(rst_n),
    .clk(clk)
    );

// Clock Scaler
  timer timer(
    .rst_n(rst_n),
    .clk(clk),
    .oneHzClk(oneHzClk),
    .ACKclk(ACKclk)
    );

// Registers (fudge it for now to test)
// (total bits: 137 = 24+64+1+16+32 for CRC32)
  register register (
    .rst_n(rst_n),
    .clk(clk),
    .uartDout(uartDout),
    .rd(rd),
    .serialShift(serialShift),
    .highCount(highCount),
    .lowCount(lowCount),
    .gndOrHiZDutyCycle(gndOrHiZDutyCycle),
    .ch1StopCnt(ch1StopCnt),
    .ch1StartCnt(ch1StartCnt),
```

```
    .ch2StopCnt(ch2StopCnt),
    .ch2StartCnt(ch2StartCnt),
    .ch3StopCnt(ch3StopCnt),
    .ch3StartCnt(ch3StartCnt),
    .ch4StopCnt(ch4StopCnt),
    .ch4StartCnt(ch4StartCnt),
    .regLoad(regLoad),
    .i2cDataOut(i2cDataOut),
    .regDataOut(regDataOut)

    );

 // for System Controller
 sysControl sysControl(
    .rst_n(rst_n),
    .clk(clk),
    .crcPass(crcPass),
    .serialShift(serialShift),
    .dataReady(dataReady),
    .uartDin(uartDin),
    .uartEn(uartEn),
    .enFreq(enFreq),
    .i2cAddr(i2cAddr),
    .i2cComplete(i2cComplete),
    .i2cRd(i2cRd),
    .i2cWr(i2cWr),
    .i2cEn(i2cEn),
    .sRegOn(sRegOn),
    .lRegOn_n(lRegOn_n),
    .lRegErr_n(lRegErr_n),
    .enChSw(enChSw),
    .enCrc(enCrc),
    .uartErr(uartErr),
    .regLoad(regLoad),
    .rd(rd),
    .i2cErr(i2cErr),
    .txComplete(txComplete),
    .oneHzClk(oneHzClk),
    .chSwComplete(chSwComplete),
    .wr(wr),
    .manErr(manErr),
    .demodErr(demodErr)

);
// CRC32
  crc crc (
     .rst_n(rst_n),
     .clk(clk),
     .serialShift(serialShift),
     .crcPass(crcPass),
     .regDataOut(regDataOut),
     .enCrc(enCrc)
);

// Frequency and PWM Generator
   freqGenerator freqGenerator(
      .rst_n(rst_n),
      .clk(clk),
      .highCount(highCount),
      .lowCount(lowCount),
      .enFreq(enFreq),
      .chBalHiPhase(chBalHiPhase),
      .fout(fout),
      .gateEnPulse(gateEnPulse));

// Channel Switcher
    chSwitcher chSwitcher (
      .rst_n(rst_n),
      .chBalHiPhase(chBalHiPhase),
```

```
    .fout(fout),
    .oneHzClk(oneHzClk),
    .clk(clk),
    .ch1StartCnt(ch1StartCnt),
    .ch1StopCnt(ch1StopCnt),
    .ch2StartCnt(ch2StartCnt),
    .ch2StopCnt(ch2StopCnt),
    .ch3StartCnt(ch3StartCnt),
    .ch3StopCnt(ch3StopCnt),
    .ch4StartCnt(ch4StartCnt),
    .ch4StopCnt(ch4StopCnt),
    .enChSw(enChSw),
    .ch1(ch1),
    .ch1_n(ch1_n),
    .ch2(ch2),
    .ch2_n(ch2_n),
    .ch3(ch3),
    .ch3_n(ch3_n),
    .ch4(ch4),
    .ch4_n(ch4_n),
    .polA(polA),
    .polB(polB),
    .gndOrHiZDutyCycle(gndOrHiZDutyCycle),
    .gateEnPulse(gateEnPulse),
    .chSwComplete(chSwComplete)
     );

// I2C Controller
  i2c i2c (
    .rst_n(rst_n),
    .clk(clk),
    .regDataOut(regDataOut),
    .i2cDataOut(i2cDataOut),
    .SDA(SDA),
    .SCL(SCL),
    .i2cAddr(i2cAddr),
    .i2cComplete(i2cComplete),
    .i2cRd(i2cRd),
    .i2cWr(i2cWr),
    .i2cEn(i2cEn),
    .serialShift(serialShift),
    .i2cErr(i2cErr),
    .ACKclk(ACKclk)
     );

// UART
  manUart manUart (
    .rxD(rxD),
    .txD(txD),
    .txComplete(txComplete),
    .uartDout(uartDout),
    .dataReady(dataReady),
    .wr(wr),
    .uartDin(uartDin),
    .uartEn(uartEn),
    .rst_n(rst_n),
    .clk(clk),
    .uartErr(uartErr),
    .manErr(manErr),
    .demodErr(demodErr)
   // .readActive(readActive)
     );


endmodule
```

A.3. Module crc.v

```
/*
Note, this module can be rewritten as a generic n-bit crc by reviewing
the design approach for a generic n-bit LFSR shown in "HDL Chip Design" by
Smith on page 185.

Also note that when feeding in the CRC value to check the data, the MSB of the
CRC must be presented first and the LSB last.  After the LSB of the received CRC
has been processed, the data is verified if the newly computed CRC is zero (i.e.
crcPass = 1).
*/

module crc(rst_n,clk,regDataOut,serialShift,enCrc,crcPass);
  input rst_n;
  input clk;
  input regDataOut;
  input serialShift;
  input enCrc;
  output crcPass;

  reg [31:0] crc;
  wire feedback;

assign crcPass = ~| crc; // reduction NOR to check if final value is 0 which
                         // means the CRC passed

//
  assign feedback = regDataOut ^ crc[31];

  always@(posedge clk)
    begin
      if (rst_n == 0)
        crc <= 0;
      else if (enCrc == 1)
        begin
          if (serialShift == 1)
            begin
              crc[0]  <= feedback;          // LSB
              crc[1]  <= feedback ^ crc[0];
              crc[2]  <= crc[1];
              crc[3]  <= crc[2];
              crc[4]  <= feedback ^ crc[3];
              crc[5]  <= crc[4];
              crc[6]  <= crc[5];
              crc[7]  <= crc[6];
              crc[8]  <= feedback ^ crc[7];
              crc[9]  <= feedback ^ crc[8];
              crc[10] <= crc[9];
              crc[11] <= feedback ^ crc[10];
              crc[12] <= feedback ^ crc[11];
              crc[13] <= feedback ^ crc[12];
              crc[14] <= feedback ^ crc[13];
              crc[15] <= feedback ^ crc[14];
              crc[16] <= crc[15];
              crc[17] <= crc[16];
              crc[18] <= feedback ^ crc[17];
              crc[19] <= feedback ^ crc[18];
              crc[20] <= crc[19];
              crc[21] <= feedback ^ crc[20];
              crc[22] <= crc[21];
              crc[23] <= feedback ^ crc[22];
              crc[24] <= crc[23];
              crc[25] <= crc[24];
              crc[26] <= feedback ^ crc[25];
              crc[27] <= crc[26];
              crc[28] <= feedback ^ crc[27];
              crc[29] <= feedback ^ crc[28];
```

```
            crc[30] <= feedback ^ crc[29];
            crc[31] <= feedback ^ crc[30]; // MSB
          end
        else
          crc <= crc;
      end
    else
      crc <= 0;
  end

endmodule
```

A.4. Module freqGenerator.v

```
// This is the state-machine version of the frequency generator

`timescale 1 ns/1 ps
module freqGenerator(rst_n,clk,highCount,lowCount,enFreq,
                     chBalHiPhase,fout,gateEnPulse);

  `include "paramFile.v"

  input rst_n;
  input clk;
  input [`pwmCountBits-1:0] highCount;
  input [`pwmCountBits-1:0] lowCount;
  input enFreq;
  output chBalHiPhase;
  output fout;
  output gateEnPulse;

  reg [`pwmCountBits:0] countTemp_r; // note that we're sharing one register to
                                     // count both the high and low phases
  reg [`pwmCountBits:0] countTemp;   // note that we're sharing one register to
                                     // count both the high and low phases

  reg chBalHiPhase, chBalHiPhase_r;
  reg [2:0] currentState,nextState;
  reg fout;

///
  reg gateEnPulse;

  always@(posedge clk)
    begin
      if (rst_n == 0)
        gateEnPulse <= 0;
      else if (countTemp == 2 && chBalHiPhase == 0 && (currentState == 5 ||
               fout == 0))
        gateEnPulse <= 1;
      else
        gateEnPulse <= 0;
    end

//

  // States
  parameter start = 0,
            getHighCount = 1,
            countHighPhase = 2,
            getLowCount = 3,
            countLowPhase = 4,
            dutyCycle100 = 5;

// combinatorial (next-state and output logic) state
  always@(currentState or countTemp_r or countTemp or enFreq or highCount or
          lowCount or chBalHiPhase_r)
    begin
      if (enFreq == 1) // enabled
        begin
          case (currentState)
            start :
              begin
                fout = 0;
                chBalHiPhase = 0;
                countTemp = highCount;
                if (lowCount == 0)
                  nextState = dutyCycle100;
                else
                  nextState = getHighCount;
```

```
  end
getHighCount :
  begin
    fout = 1;
    chBalHiPhase = !chBalHiPhase_r; // toggle chBalHiPhase to
                                    // preserve charge balance
                                    // information
    countTemp=highCount;
    if (countTemp == 1)
      nextState = getLowCount;
    else
      nextState = countHighPhase;
  end
countHighPhase :
  begin
    fout = 1;
    chBalHiPhase = chBalHiPhase_r;
    countTemp = countTemp_r;
    if (countTemp == 1)
      nextState = getLowCount;
    else
      nextState = countHighPhase;
  end
getLowCount :
  begin
    fout = 0;
    chBalHiPhase = chBalHiPhase_r;
    countTemp=lowCount;
    if (countTemp == 1)
      nextState = getHighCount;
    else
      nextState = countLowPhase;
  end
countLowPhase :
  begin
    fout = 0;
    chBalHiPhase = chBalHiPhase_r;
    countTemp = countTemp_r;
    if (countTemp == 1)
      nextState = getHighCount;
    else
      nextState = countLowPhase;
  end
dutyCycle100 :
  begin
    fout = 1;
    if (countTemp_r == 0)
      begin
        countTemp = highCount;
        chBalHiPhase = !chBalHiPhase_r;
        if (lowCount != 0)
          nextState = start;
        else
          nextState = dutyCycle100;
      end
    else
      begin
        countTemp = countTemp_r;
        chBalHiPhase = chBalHiPhase_r;
        nextState = dutyCycle100;
      end
  end
default : // should never get here
  begin
    fout = 0;
    chBalHiPhase = 0;
    countTemp = 0;
    nextState = start;
  end
```

```
            endcase
          end
        else // not enabled
          begin
            fout = 0;
            chBalHiPhase = 0;
            countTemp = 0;
            nextState = start;
          end
      end

  // sequential state
    always@(posedge clk)
      begin
        if (rst_n == 0)
          begin
            currentState <= start;
            countTemp_r <= 0;
            chBalHiPhase_r <= 0;
          end
        else
          begin
            currentState <= nextState;
            countTemp_r <= countTemp - 1;
            chBalHiPhase_r <= chBalHiPhase;
          end
      end

  endmodule
```

A.5. Module i2c.c

```
module i2c(rst_n,clk,SDA,SCL,i2cAddr,i2cComplete,i2cRd,i2cWr,i2cEn,regDataOut,
          i2cDataOut,serialShift,i2cErr,ACKclk);
// Ports

    input  rst_n;
    input  clk;
    output SCL;
    inout  SDA;           // note, type inout
    input  [15:0] i2cAddr;// i2c address plus a control or address byte:
                          // for the serial EEPROM this will contain the lower
                          // 4 bits of the memory address (only 16 bytes to be
                          // addressed), e.g. XXXX0101,
                          // for the digital potentiometer this will contain the
                          // instruction byte, e.g. X00XXXXX
    output i2cComplete;
    input  i2cRd;         // 1 clk pulse
    input  i2cWr;         // 1 clk pulse
    input  i2cEn;         // static
    input  regDataOut;
    output i2cDataOut;
    input  serialShift;   // one shift for every clk cycle this signal is high
    output i2cErr;
    input ACKclk;

// Signals

  reg [7:0] i2cReg;   // internal data register for i2c data
  reg i2cSerialShift; // internal control signal to shift in data from SDA_out (for i2c
reads)
  reg [4:0] currentState;
  reg [4:0] nextState;
  wire [7:0] i2cSlaveAddr;
  wire [7:0] i2cControlByte;
  reg [3:0] bytePointer;
  reg readTransfer;
  //wire SCL;
  reg i2cErr;
  reg i2cComplete;
  reg SDA_out;
  wire SDA_in;
  reg SDA_oe; // enabled when 1, hiZ when 0
  reg [1:0] sendByteState; // Holds the state for write transfers (to enable
                           // switch between bytes to be sent, to reuse the
                           // send state machine). States: address, control,
                           // and wrData.

  reg [1:0] sendByteState_r;
  reg SDA_oe_r;
  reg SDA_out_r;
  reg SCL_r;
//  reg i2cErr_r;
  reg [3:0] bytePointer_r;
//  reg i2cComplete_r;
  reg i2cSerialShift_r;
  reg readTransfer_r;
  wire spromAccess;
  reg SCL_int;
  reg bypass,bypass_r;
  reg [1:0] ACKclkCnt;
  reg ACKclkCntClear;
  reg reachedDefault;

  assign spromAccess = i2cSlaveAddr[7]; // if 1 then accessing the serial
                                        // EEPROM, otherwise if 0, then
                                        // accessing the programmable
                                        // voltage regulators
```

```
  assign i2cSlaveAddr[7:1] = i2cAddr[15:9];

  // this conditional statement below controls the value of the r/w_n bit in the i2c
address,
  // ordinarily, the sysControl module will set this bit but during a random i2c read
  // of the sprom, a write then a read i2c operation occurs
//  always@(spromAccess or readTransfer_r or bypass_r or i2cAddr[8])
//    begin
//      if (
//    end

  assign i2cSlaveAddr[0] = ((spromAccess == 1 && readTransfer_r == 1 &&
                             bypass_r == 0) ? !i2cAddr[8] : i2cAddr[8]);//i2cAddr[8] :
!i2cAddr[8]);

  assign i2cControlByte = i2cAddr[7:0];

  // Send Byte States
  parameter address = 0,
            control = 1,
            wrData  = 2
            ;

  // i2c State Machine States
  parameter idleS0            =  0,
            startBitS1        =  1,
            startBitS2        =  2,
            sendI2cAddrS3     =  3,
            sendI2cAddrS4     =  4,
            sendI2cAddrS5     =  5,
            sendI2cAddrEndS6  =  6,
            checkRdAckS7      =  7,
            checkRdAckS8      =  8,
            readI2cS9         =  9,
            readI2cS10        = 10,
            readI2cS11        = 11,
            readI2cEndS12     = 12,
            genStopBitS13     = 13,
            genStopBitS14     = 14,
            genStopBitS15     = 15,
            genStopBitS16     = 16,
            spromReadStartS17 = 17,
            spromReadStartS18 = 18
            ;

  assign SDA_in = SDA;
  assign SDA = (SDA_oe == 1 ? SDA_out_r : 1'bz);
  assign SCL = SCL_r;

  assign i2cDataOut = i2cReg[7];

// check for i2c error
  always@(posedge clk)
    begin
      if (rst_n == 0 )
        i2cErr = 0;
      else if ((currentState == checkRdAckS8 && SDA == 1) ||
                reachedDefault == 1) // then error
        i2cErr = 1;
       //  i2cErr = 0;
      else
        i2cErr = 0;
    end

// i2c Data Register
  always@(posedge clk)
    begin
      if (rst_n == 0 )
```

```
      i2cReg <= 0;
    else if (i2cEn == 0)
      i2cReg <= 0;
    else if (serialShift == 1) // then bit into LSB of shift register
      i2cReg <= {i2cReg[6:0],regDataOut}; // bit comes in from register.v
    else if (i2cSerialShift == 1) // then bit into LSB shift register
      i2cReg <= {i2cReg[6:0],SDA_in}; // bit comes in from i2c bus
    else
      i2cReg <= i2cReg;
  end

// counter to check i2c ACK bit timeout for sprom writes (max 4ms delay)
  always@(posedge clk)
    begin
      if (rst_n == 0 )
        ACKclkCnt <= 0;
      else if (ACKclkCntClear == 1)
         ACKclkCnt <= 0;
      else if (ACKclk == 1)
         ACKclkCnt <= ACKclkCnt + 1;
    end

// i2c State Machine

  always@(posedge clk)
    begin
      if (rst_n == 0 )
      begin
            sendByteState_r = address;
            SDA_oe_r = 1;
            SDA_out_r = 1;
            SCL_r = 1;
            bytePointer_r = 4'b 0111;
            i2cSerialShift_r = 0;
            readTransfer_r = 0;
            bypass_r = 0;
    end
    else
      begin
            sendByteState_r = sendByteState;
            SDA_oe_r = SDA_oe;
            SDA_out_r = SDA_out;
            SCL_r = SCL_int;
            bytePointer_r = bytePointer;
            i2cSerialShift_r = i2cSerialShift;
            readTransfer_r = readTransfer;
            bypass_r = bypass;
     end
    end
// combinatorial
  always@(currentState      or
          i2cRd             or
          i2cWr             or
          readTransfer      or
          i2cSlaveAddr      or
          SDA_out           or
          bytePointer       or
          sendByteState     or
          i2cControlByte    or
          i2cReg            or
          SDA               or
          sendByteState_r   or
          SDA_oe_r          or
          SDA_out_r         or
          SCL_r             or
          bytePointer_r     or
          i2cSerialShift_r  or
          readTransfer_r    or
          spromAccess     or
```

```
            bypass              or
            bypass_r            or
            ACKclkCnt
        )

       begin
           sendByteState = sendByteState_r;
           SDA_oe = SDA_oe_r;
           SDA_out = SDA_out_r;
           SCL_int = SCL_r;
           bytePointer = bytePointer_r;
           i2cComplete = 0; //debug i2cComplete_r;
           i2cSerialShift = i2cSerialShift_r;
           readTransfer = readTransfer_r;
           bypass = bypass_r;
           ACKclkCntClear = 0;
           reachedDefault = 0;
     case (currentState)

       idleS0 :
         begin
           SDA_out = 1;
           SCL_int = 1;
           SDA_oe = 1;
           sendByteState = 0;
           bytePointer = 4'b 0111;
           i2cComplete = 1;
           bypass = 0;
           if (i2cRd == 1)
             begin
               nextState = startBitS1;
               readTransfer = 1;
             end
           else if (i2cWr == 1)
             begin
               nextState = startBitS1;
               readTransfer = 0;
             end
           else
             begin
               nextState = idleS0;
               readTransfer = readTransfer_r;
             end
         end

// i2c start bit generation

        spromReadStartS17 :
          begin
              sendByteState = 0;
              bytePointer = 4'b 0111;
              SDA_oe = 1;
              SDA_out = 1;
              SCL_int = 0;
              nextState = spromReadStartS18;
          end

        spromReadStartS18 :
          begin
              sendByteState = 0;
              bytePointer = 4'b 0111;
              SDA_oe = 1;
              SDA_out = 1;
              SCL_int = 1;
              nextState = startBitS1;
          end

        startBitS1 :
          begin
```

```
            SDA_oe = 1;
            SDA_out = 0;
            SCL_int = 1;
            nextState = startBitS2;
          end

        startBitS2 :
          begin
            SDA_out = 0;
            SCL_int = 0;
            nextState = sendI2cAddrS3;
          end

// send i2c address

        sendI2cAddrS3 :
          begin
            SCL_int = 0;
            SDA_oe = 1; // make SDA output
            if (bytePointer == 4'b 1111)
              begin
                SDA_oe = 0; // make SDA hiZ
                SDA_out = SDA_out_r;
                nextState = sendI2cAddrEndS6;
              end
            else
              begin
                nextState = sendI2cAddrS4;
                case (sendByteState)
                  address : SDA_out = i2cSlaveAddr[bytePointer];
                  control : SDA_out = i2cControlByte[bytePointer];
                  wrData  : SDA_out = i2cReg[bytePointer];
                  default : SDA_out = 1; // should never get here
                endcase
              end
          end

        sendI2cAddrS4 :
          begin
            SDA_out = SDA_out_r;
            SCL_int = 1;
            bytePointer = bytePointer_r - 1;
            nextState = sendI2cAddrS5;
          end

        sendI2cAddrS5 : // can only change SDA when SCL_int is low, this prepares for the
next state
          begin
            SDA_out = SDA_out_r;
            SCL_int = 0;
            nextState = sendI2cAddrS3;
          end

        sendI2cAddrEndS6 : //Note that i2cSlaveAddr[0] must be 1 by the sysControl
                           //block to indicate a read
          begin
            SDA_out = SDA_out_r;
            SCL_int = 0;
            nextState = checkRdAckS7;
          end

// check acknowledgement

        checkRdAckS7 :
          begin
            SCL_int = 1;
            nextState = checkRdAckS8;
          end
```

```
        checkRdAckS8 :
          begin
          //  i2cErr = 0;
            SCL_int = 0;
//            if (SDA == 0) // then passed acknowledgement
//              begin
                if (readTransfer == 1 && (spromAccess == 0 || bypass == 1)) // then read
the Vreg or second i2c operation for sprom read
                  begin
                    bytePointer = bytePointer_r;
                    sendByteState = sendByteState_r;
                   // i2cErr = i2cErr_r;

                    nextState = readI2cS9;
                  end
                else
                  case (sendByteState_r)
                    address : begin // ready to write control word for write to Vreg or
address for sPROM
                      bytePointer = 4'b 0111;
                      sendByteState = control;
                     // i2cErr = i2cErr_r;
                      nextState = sendI2cAddrS3; end
                    control :
                      begin // ready to write data byte
                        bytePointer = 4'b 0111;
                        sendByteState = wrData;
                        if (readTransfer == 1 && spromAccess == 1) // if accessing the
sPROM for read, then go to the read rountine
                          begin
                      //      nextState = readI2cS9;
                            nextState = spromReadStartS17;
                            bypass = 1;
                          end
                        else
                          begin
                            nextState = sendI2cAddrS3;
                          end
                      end
                    wrData  : begin
                      bytePointer = 4'b 0111;
                      sendByteState = address;
                    //  i2cErr = i2cErr_r;
                      nextState = genStopBitS13; end
                    default : begin // should never get here
                      bytePointer = bytePointer_r;
                      sendByteState = address;
                    //  i2cErr = 1;
                      nextState = genStopBitS13; end
                  endcase
              end

// read

        readI2cS9 :
          begin
            bytePointer = 4'b 0111;
            SCL_int = 1;
            nextState = readI2cS10;
          end

        readI2cS10 :
          begin
            SCL_int = 0;
            if (bytePointer == 4'b 1111) // then
              begin
                nextState = genStopBitS13; // was readI2cEndS12
              end
            else
```

```
            begin
              i2cSerialShift = 1;
              nextState = readI2cS11;
            end
        end

      readI2cS11 :
        begin
          SCL_int = 1;
          i2cSerialShift = 0;
          bytePointer = bytePointer_r - 1;
          nextState = readI2cS10;
        end

      readI2cEndS12 : //(bit 9)
        begin
          SCL_int = 1;
          nextState = genStopBitS13;
        end

// stop bit
      genStopBitS13 :
        begin
          SDA_out = 0;
          SCL_int = 0;
          nextState = genStopBitS14;
        end

      genStopBitS14 :
        begin
          SDA_oe = 1;
          nextState = genStopBitS15;
        end

      genStopBitS15 :
        begin
          SCL_int = 1;
          nextState = genStopBitS16;
          ACKclkCntClear = 1;
        end

      genStopBitS16 :
        begin
          SDA_out = 1; // complete the stop condition
          if (readTransfer == 0 && spromAccess == 1) //then it's a sprom write, need a
delay
            begin
              if (ACKclkCnt == 2'b 10)// wait for time delay to elapse before
continuing
                begin
                  i2cComplete = 1;
                  nextState = idleS0;
                end
              else
                nextState = genStopBitS16;
            end
          else
            begin
              i2cComplete = 1;
              nextState = idleS0;
            end
        end

      default : // should never get here
        begin
          SDA_out = 1;
          SCL_int = 1;
          reachedDefault = 1;
          nextState = genStopBitS13;
```

```
          end
        endcase
      end

// sequential state
  always@(posedge clk)
    begin
      if (rst_n == 0)
        currentState <= idleS0;
    else if (i2cEn == 0)
        currentState <= idleS0;
    else
        currentState <= nextState;
     end

endmodule
```

A.6. Module manUart.v

```
// Manchester UART.  Transmit is same as regular UART, receive combines byte
// into 4 nibbles so you need two receive bytes to make a byte.

// NEED TO INVERT INPUT SIGNAL TO CONFORM TO JAMES CICRCUIT
// NEED TO CHANGE OPERATING RATE FROM 1200BPS TO 2400BPS

/*
The correct method to download data via the PC is to:
1. convert the 144-bit binary string to a 288-bit Manchester encoded binary string
2. convert the 288-bit binary string to ASCII, then
3. save the ASCII to a text file, then
4. open Microsoft HyperTerminal and in "Properties" select COM1 and under "Configure"
   select:
          Bits per second: 2400
                Data bits: 8
                   Parity: None
                Stop bits: 2
            Flow Control: None
5. use HyperTerminal's "Send Text File" under the "Transfer" drop down menu,
   select the text file created in step 3.


*/

`timescale 1 ns/1 ps

module manUart(rxD,txD,txComplete,uartDout,dataReady,wr,uartDin,uartEn,rst_n,clk,
            uartErr,manErr,demodErr//,readActive
            );

// Ports

  input  rxD; // uart serial input
  output txD; // uart serial output
  output  txComplete;
  output [7:0] uartDout; // parallel output of received byte
  output dataReady; // strobe that indicates new data available that hasn't been read
  input  wr; // write strobe to uartDin, initiates a write out on txD (if rxd not
active), no need to check if it's done because we'll wait for a timeout to receive a
command
  input  [7:0] uartDin; // data to be sent from the uart through txD, strobed in by wr
  input  uartEn; //enable
  input  rst_n;
  input  clk;
  output uartErr; //stop bit error on character reception
  output manErr; // manchester error
  output demodErr;//demodulator error (signal always in Start bit state

// Signals

reg [1:0] sampleRegR;
wire [2:0] sampleReg;
reg rxD1,rxD_r; // double registered (synchronized) rxD input
//reg rxD_m;  // majority voting output
reg rxD_mr; // registered version of majority voting output
reg [4:0] clockGenReg;
reg clockGenPulse;
reg [1:0] clockCount;
reg startClkGen; // high when a potential start bit is detected and low when
                 // start bit is found false or when the stop bit is detected
reg firstHalfBit; // flag to mark counting the first half bit for checking the
                  // stop bit and creating the start point for the bit sampling
reg firstHalfBitComplete;
reg [7:0] rxReg;   // internal data register for uart receive data
reg serialRxShift;
reg [2:0] bitPointer; // for receive shift register
```

```
reg [3:0] currentState;
reg [3:0] nextState;

reg dataReady;
reg uartErr;
reg manErr;
reg demodErr;
reg txD_c;
reg txD_next,txD_current;
reg txD_i;
reg txComplete;
// can lose the msb of nibbleStatus without any loss of functionality
reg [1:0] nibbleStatus; // 01 = first byte/nibble done, 10 = second byte/nibble done
reg nibbleDone;
reg [7:0] rxRegTemp;
wire manErrFlag;

// maybe more efficient to do the Manchester decode and error check with a
// sliding 2 register circuit, maybe
//Manchester decode, note it is simply done, 11 and 00 are flagged as errors later

//correct the manchester decoding (invert, swap nibbles, reverse bit order)
assign uartDout[0] =      (rxReg[7:6] == 2'b 10 ? 0 : 1);
assign uartDout[1] =      (rxReg[5:4] == 2'b 10 ? 0 : 1);
assign uartDout[2] =      (rxReg[3:2] == 2'b 10 ? 0 : 1);
assign uartDout[3] =      (rxReg[1:0] == 2'b 10 ? 0 : 1);
assign uartDout[4] = (rxRegTemp[7:6] == 2'b 10 ? 0 : 1);
assign uartDout[5] = (rxRegTemp[5:4] == 2'b 10 ? 0 : 1);
assign uartDout[6] = (rxRegTemp[3:2] == 2'b 10 ? 0 : 1);
assign uartDout[7] = (rxRegTemp[1:0] == 2'b 10 ? 0 : 1);

// error flag for Manchester decode
assign manErrFlag = (rxRegTemp[7] ~^ rxRegTemp[6]) ||
                    (rxRegTemp[5] ~^ rxRegTemp[4]) ||
                    (rxRegTemp[3] ~^ rxRegTemp[2]) ||
                    (rxRegTemp[1] ~^ rxRegTemp[0]) ||
                        (rxReg[7] ~^     rxReg[6]) ||
                        (rxReg[5] ~^     rxReg[4]) ||
                        (rxReg[3] ~^     rxReg[2]) ||
                        (rxReg[1] ~^     rxReg[0]) ;


//
  always@(posedge clk)
    begin
      if (rst_n == 0 )
        begin
          nibbleStatus  <= 0;//rxD; // prevent a glitch in the beginning
          rxRegTemp     <= 0; // Denis synplify
        end
      else if (dataReady == 1)//(nibbleStatus == 2'b 10) debug
        begin
          nibbleStatus  <= 0;
        end
      else if (nibbleDone == 1)
        begin
          nibbleStatus  <= nibbleStatus + 1;
          rxRegTemp     <= rxReg;
        end
    end

// synchronizer for asynchronous input
  always@(posedge clk)
    begin
      if (rst_n == 0 ) begin
        rxD1  <= 1;//rxD; // prevent a glitch in the beginning
        rxD_r <= 1;//rxD;
        end
      else begin
```

```
        rxD1  <= rxD;
        rxD_r <= rxD1;
        end
    end

// rxD Sampler - continuously takes 3 samples and determines the majority value for
glitch rejection
  always@(posedge clk)
    begin
      if (rst_n == 0 )
        sampleRegR <= 0; // prevent a glitch in the beginning //Denis synplify
      else
        sampleRegR <= {sampleRegR[0],rxD_r}; // serial shift register
    end

assign sampleReg = {sampleRegR,rxD_r};

// majority voting for input, can easily make this a 5-bit majority voting
  always@(sampleReg)
    begin
      rxD_mr = 0; //rxD_mr = 1;
      case (sampleReg)
        3'b 000 : rxD_mr = 1;//rxD_mr = 0;
        3'b 001 : rxD_mr = 1;//rxD_mr = 0;
        3'b 010 : rxD_mr = 1;//rxD_mr = 0;
        3'b 100 : rxD_mr = 1;//rxD_mr = 0;
        default : ;
      endcase
    end

// clock generator to convert the system clock to a uart compatible time base
// i.e. 1200 or 2400 bps

// For 2400 bps
// 32768/1200 = 27.3066, so count 27 twice then 28 once then repeat to get
// (27 + 27 + 28)/3 = 27.333

// For 2400 bps
// 32768/2400 = 13.65, so count 14 twice then 13 once then repeat to get
// (14 + 14 + 13)/3 = 13.666

// For 9600 bps
// 32768/9600 = 3.4133, so count 3 twice then 4 once then repeat to get
// (3 + 3 + 4)/3 = 3.333


  always@(posedge clk)
    begin
      if (rst_n == 0)
        begin
          clockGenReg   <= 0;
          clockCount    <= 0;
          clockGenPulse <= 0;
          firstHalfBit  <= 0;
          firstHalfBitComplete <= 0;
        end
      else if (startClkGen == 0) // not enabled state
        begin
          clockGenReg   <= 0;
          clockCount    <= 0;
          clockGenPulse <= 0;
          firstHalfBit  <= 0;
          firstHalfBitComplete <= 0;
        end
      else if (clockGenReg == 0 && firstHalfBit == 0 && firstHalfBitComplete == 0) //
start count for first half bit
        firstHalfBit <= 1;
//      else if (clockGenReg >= 12 && firstHalfBit == 1) // end count for first half bit
("middle" of bit)//for 1200bps
```

```
      else if (clockGenReg >= 5  && firstHalfBit == 1) // end count for first half bit
("middle" of bit)//for 2400bps
        begin
          clockGenReg   <= 0;
          firstHalfBit  <= 0;
          clockGenPulse <= 1;
          firstHalfBitComplete <= 1;
        end
//      else if (clockGenReg >= 26 && clockCount < 2 && firstHalfBit == 0) // count 27
cycles twice for bit sampling//for 1200bps
      else if (clockGenReg >= 13 && clockCount < 2 && firstHalfBit == 0) // count 14
cycles twice for bit sampling//for 1200bps
        begin
          clockGenReg   <= 0;
          clockCount    <= clockCount + 1;
          clockGenPulse <= 1;
        end
//      else if (clockGenReg >= 27 && clockCount >= 2 && firstHalfBit == 0) // count 28
cycles once for bit sampling//for 1200bps
      else if (clockGenReg >= 12 && clockCount >= 2 && firstHalfBit == 0) // count 13
cycles once for bit sampling//for 1200bps
        begin
          clockGenReg   <= 0;
          clockCount    <= 0;
          clockGenPulse <= 1;
        end
      else
        begin
          clockGenReg   <= clockGenReg + 1;
          clockGenPulse <= 0;
        end
    end

// uartRx Data Register
  always@(posedge clk)
    begin
      if (rst_n == 0 )
        begin
          rxReg <= 0;
          bitPointer <= 7;
        end
      else if (serialRxShift == 1) // then bit into LSB of shift register
        begin
          rxReg <= {rxReg[6:0],rxD_mr};
          bitPointer <= bitPointer - 1;
        end
      else if (startClkGen == 0)
        begin
          rxReg <= rxReg;
          bitPointer <= 7;
        end
      else
        begin
          rxReg <= rxReg;
          bitPointer <= bitPointer;
        end
    end

// State Machine to control the clock generator signals
// outputs: startClkGen, uartErr
// inputs : rxD_mr, clockGenPulse

  // uart State Machine States
  parameter idleS0           =  0,
            checkStartBitS1  =  1,
            getByteS2        =  2,
            checkStopBitS3   =  3,
            waitForIdleS4    =  4,
            genStartBitS5    =  5,
```

```
            genByteS6        =  6,
            genStopBitS7     =  7,
            genStopBitS8     =  8
            ;

// combinatorial
  always@(currentState      or
          uartEn            or
          rxD_mr            or
          clockGenPulse     or
          bitPointer        or
          wr                or
          txD_current       or
          uartDin           or
          manErrFlag        or
          nibbleStatus
         )
    begin
      // default statements
      startClkGen = 1;
      serialRxShift = 0;
      dataReady = 0;
      uartErr = 0;
      manErr = 0;
      demodErr = 0;
      nextState = idleS0;
      // readActive = 0;
      txD_c = 1;
      txD_next = 1;
      txComplete = 1;
      nibbleDone = 0;

      if (uartEn == 1)
        begin
          case (currentState)

            idleS0 :
              begin
                if (wr == 1) // command to send a byte out through the uart
                  begin
                    startClkGen = 1;
                    nextState = genStartBitS5;
                    txComplete = 0;
                  end
                else if (rxD_mr == 0) // found a potential start bit
                  begin
                    startClkGen = 1;
                    nextState = checkStartBitS1;
                    // readActive = 1;
                  end
                else
                  begin
                    startClkGen = 0;
                    nextState = idleS0;
                    // readActive = 0;
                  end
              end

 // Read uart operation
            checkStartBitS1 :
              begin
                if (clockGenPulse == 1) // check a potential start bit at midpoint
                  begin
                    if (rxD_mr == 0) // it is a start bit
                      begin
                        startClkGen = 1;
                        nextState = getByteS2;
                        // readActive = 1;
                      end
```

```
            else // not a start bit, go back to the beginning
              begin
                startClkGen = 0;
                nextState = idleS0;
                // readActive = 0;
              end
          end
        else // wait for clockGenPulse to indicate the bit sampling time
          begin
            startClkGen = 1;
            nextState = checkStartBitS1;
            // readActive = 1;
          end
      end

    getByteS2 :
      begin
      // readActive = 1;
      startClkGen = 1;
        if (clockGenPulse == 1) // get the receive bits at midpoints
          begin
            serialRxShift = 1; // shift in the received bit
            if (bitPointer == 0)
              nextState = checkStopBitS3;
            else
              nextState = getByteS2; // get next bit
          end
        else // wait for clockGenPulse to indicate the bit sampling time
          begin
            nextState = getByteS2;
          end
      end

    checkStopBitS3 :
      begin
      // readActive = 1;
      startClkGen = 1;
        if (clockGenPulse == 1) // get the receive bits at midpoints
          begin
            nextState = idleS0;
            if (rxD_mr == 1) // check if stop bit is detected, STOP BIT is a 1,
not 0
              begin
                nextState = waitForIdleS4;
                if (nibbleStatus == 2'b 00) // only load top nibble reg
                                            // with first nibble
                    nibbleDone = 1;
                  if (nibbleStatus == 2'b 01) //(nibbleStatus == 2'b 01) debug
                  // only provide dataReady signal when 2nd nibble ready
                    dataReady = 1;
                  if (manErrFlag == 1 && nibbleStatus == 2'b 01)
                  // only flag Manchester error if it's the second nibble

                    manErr = 1; // Manchester error detected
                end
              else
                begin
                  uartErr = 1; // stop bit error detected
                  nextState = idleS0;
                end
            end
          else // wait for clockGenPulse to indicate the bit sampling time
            begin
              nextState = checkStopBitS3;
            end
        end

      waitForIdleS4 :
        begin
```

```
                if (clockGenPulse == 1 && rxD_mr == 1) // wait for rxD to go high
again
                  begin
                    startClkGen = 0;
                    nextState = idleS0;
                    // readActive = 0;
                  end
                else // wait for clockGenPulse to indicate the bit sampling time
                  begin
                    demodErr  = 1; // stop bit error detected
                    nextState = idleS0;
                  end
              end

// Write uart operation
            genStartBitS5 :
              begin
                txComplete = 0;
                startClkGen = 1;
                if (clockGenPulse == 1)
                  begin
                    txD_c = 0;
                    txD_next = 0;
                    nextState = genByteS6;
                  end
                else
                  begin
                    txD_c = 1;
                    nextState = genStartBitS5;
                  end
              end

              genByteS6 :
              begin
                txComplete = 0;
                startClkGen = 1;
                txD_c = txD_current;
                if (clockGenPulse == 1)
                  begin
                    serialRxShift = 1;// borrow the receiver's bit counter to point to
the desired bit
                    txD_c = uartDin[bitPointer];
                    txD_next = uartDin[bitPointer];
                    if (bitPointer == 0)
                      nextState = genStopBitS7;
                    else
                      nextState = genByteS6;
                  end
                else
                  begin
                    nextState = genByteS6;
                  end
              end

            genStopBitS7 :
              begin
                txComplete = 0;
                startClkGen = 1;
                txD_c = txD_current;
                if (clockGenPulse == 1)
                  begin
                    txD_c = 1;//STOP bit is 1
                    txD_next = 1;
                    nextState = genStopBitS8;
                  end
                else
                  begin
                    nextState = genStopBitS7;
                  end
```

```
            end

          genStopBitS8 :
            begin
              txComplete = 0;
              startClkGen = 1;
              txD_c = txD_current;
              if (clockGenPulse == 1)
                begin
                  txD_c = 1; //STOP bit is 1
                  txD_next = 1;

                  nextState = idleS0;
                end
              else
                begin
                  nextState = genStopBitS8;
                end
            end

          default : ;
        endcase
      end
    else // default outputs while not enabled (uartEn = 0)
      begin
        begin
          startClkGen = 0;
          nextState = idleS0;
          txComplete   = 1;
        end
    end

// sequential state
  always@(posedge clk)
    begin
      if (rst_n == 0)
        begin
          txD_i        <= 1;
          currentState <= idleS0;
        end
      else if (uartEn == 0)
        begin
          txD_i        <= 1;
          currentState <= idleS0;
        end
      else
        begin
          txD_i        <= txD_c; // clean up txD from combinatorial glitches
          currentState <= nextState;
        end
    end

//invert TxD signal to comply with transcutaneous power and data link polarity
assign txD = !txD_i;

  always@(posedge clk)
    begin
      if (rst_n == 0)
        begin
          txD_current  <= 1;
        end
      else if (uartEn == 0)
        begin
          txD_current  <= 1;
        end
      else  if (clockGenPulse == 1)
        begin
          txD_current  <= txD_next;
        end
    end
```

```
endmodule
```

A.7. Module chSwitcher.v

```
/*
Note:
1. We want all analog switches to be inactive at reset,
2. when stimulating, the relevant analog switches to the
   inactive channel(s) will be inactive, except the polA
   signal must still be active,
3. if all channels are inactive then all analog switches
   must be inactive.

Module Description:
===================
This module gates the "fout" signal to the electrodes. The "fout" signal is
timed from the 32.768kHz clock.  The channel switching is defined by the
design requirements on a one second scale, however, to preserve charge balance,
the gating of the "fout" signal must be timed from the 32.768kHz clock.  For
each channel, gating must start at the positive edge of the positive pulse and
end at the postive edge of the negative pulse to ensure charge balance.

*/

`timescale 1 ns/1 ps

module chSwitcher(rst_n,chBalHiPhase,fout,oneHzClk,clk,
                  ch1StartCnt,ch1StopCnt,ch2StartCnt,ch2StopCnt,ch3StartCnt,
                  ch3StopCnt,ch4StartCnt,ch4StopCnt,//pauseAfterStimCnt,
                  enChSw,ch1,ch1_n,ch2,ch2_n,ch3,ch3_n,ch4,ch4_n,polA,polB,
                  gndOrHiZDutyCycle,gateEnPulse,chSwComplete);

  `include "paramFile.v"

  input rst_n;
  input chBalHiPhase;
  input fout;
  input oneHzClk;
  input clk;

  input [`chSwTimerBits-1:0] ch1StartCnt;
  input [`chSwTimerBits-1:0] ch1StopCnt;
  input [`chSwTimerBits-1:0] ch2StartCnt;
  input [`chSwTimerBits-1:0] ch2StopCnt;
  input [`chSwTimerBits-1:0] ch3StartCnt;
  input [`chSwTimerBits-1:0] ch3StopCnt;
  input [`chSwTimerBits-1:0] ch4StartCnt;
  input [`chSwTimerBits-1:0] ch4StopCnt;

//  input [`chSwTimerBits-1:0] pauseAfterStimCnt;
  input enChSw;
  input gndOrHiZDutyCycle;
  input gateEnPulse;

  output ch1;   // 1 is active, 0 inactive
  output ch1_n; // 0 is active, 1 inactive
  output ch2;   // 1 is active, 0 inactive
  output ch2_n; // 0 is active, 1 inactive
  output ch3;   // 1 is active, 0 inactive
  output ch3_n; // 0 is active, 1 inactive
  output ch4;   // 1 is active, 0 inactive
  output ch4_n; // 0 is active, 1 inactive
  output polA;  // 1 is active, 0 inactive
  output polB;  // 0 is active, 1 inactive
  output chSwComplete ; // 0 incomplete channel stimulation, 1 complete

  reg [`chSwTimerBits-1:0] chSwTimer;

//  reg stimSessComplete;
```

```
reg ch1_i;
reg ch2_i;
reg ch3_i;
reg ch4_i;

wire ch1_n_i;
wire ch2_n_i;
wire ch3_n_i;
wire ch4_n_i;

reg ch1;
reg ch1_n;
reg ch2;
reg ch2_n;
reg ch3;
reg ch3_n;
reg ch4;
reg ch4_n;
reg polA;
reg polB;

reg ch1temp;
reg ch2temp;
reg ch3temp;
reg ch4temp;
reg polA_i;
reg polB_i;

reg gateCh1;
reg gateCh2;
reg gateCh3;
reg gateCh4;

reg currentState1;
reg currentState2;
reg currentState3;
reg currentState4;

reg nextState1;
reg nextState2;
reg nextState3;
reg nextState4;

reg oneHzClk_r;

reg chSwComplete;
reg ch1Complete,ch2Complete,ch3Complete,ch4Complete;

wire polEn;
wire oneHzPulse;


// States
parameter idle = 0, active = 1;

// Timer (I fixed this so everything would be on a single clock domain)
always@(posedge clk)
  begin
    if (rst_n == 0 )
      oneHzClk_r <= 0;
    else
      oneHzClk_r <= oneHzClk;
  end

assign oneHzPulse = (oneHzClk ^ oneHzClk_r) && oneHzClk;

// note timer will roll over so the programmed seconds have to be -1 the desired
// number of seconds
```

```
  always@(posedge clk)
    begin
      if (rst_n == 0 )
        chSwTimer <= 8'b 1111_1111;
      else if (enChSw == 0)
        chSwTimer <= 8'b 1111_1111;
      else if (oneHzPulse == 1)
        chSwTimer <= chSwTimer + 1;
      else
        chSwTimer <= chSwTimer; // can remove this line because this is synchronous
process
    end

// ch1temp Timing Comparator
  always@(posedge clk)
    begin
      if (rst_n == 0)
        begin
          gateCh1     <= 0;
          ch1Complete <= 0;
        end
      else if (chSwTimer == ch1StopCnt)
        begin
          gateCh1     <= 0;
          ch1Complete <= 1;
        end
      else if (ch1StartCnt == chSwTimer)
        gateCh1 <= 1;
    end

// ch2temp Timing Comparator
  always@(posedge clk)
    begin
      if (rst_n == 0)
        begin
          gateCh2     <= 0;
          ch2Complete <= 0;
        end
      else if (chSwTimer == ch2StopCnt)
        begin
          gateCh2     <= 0;
          ch2Complete <= 1;
        end
      else if (ch2StartCnt == chSwTimer)
        gateCh2 <= 1;
    end

// ch3temp Timing Comparator
  always@(posedge clk)
    begin
      if (rst_n == 0)
        begin
          gateCh3     <= 0;
          ch3Complete <= 0;
        end
      else if (chSwTimer == ch3StopCnt)
        begin
          gateCh3     <= 0;
          ch3Complete <= 1;
        end
      else if (ch3StartCnt == chSwTimer)
        gateCh3 <= 1;
    end

// ch4temp Timing Comparator
  always@(posedge clk)
    begin
      if (rst_n == 0)
        begin
```

```
        gateCh4     <= 0;
        ch4Complete <= 0;
      end
    else if (chSwTimer == ch4StopCnt)
      begin
        gateCh4     <= 0;
        ch4Complete <= 1;
      end
    else if (ch4StartCnt == chSwTimer)
      gateCh4 <= 1;
  end

  always@(ch1Complete or ch2Complete or ch3Complete or ch4Complete)
// too big to fit    chSwComplete = ch1Complete & ch2Complete & ch3Complete & ch4Complete
& !polEn;
  chSwComplete = ch1Complete & ch4Complete & !polEn & !chBalHiPhase;

  assign polEn = ch1_i | ch2_i | ch3_i | ch4_i;

// Control polarity of signal
  always@(fout or chBalHiPhase or enChSw or polEn)
    begin
      if (enChSw == 0 || fout == 0 || polEn ==0) // then ground both
        begin
          polA_i = 0;
          polB_i = 1;
        end
      else if (chBalHiPhase == 1) // then avdd and gnd
        begin
          polA_i = 1;
          polB_i = 1;
        end
      else  // then gnd and avdd
        begin
          polA_i = 0;
          polB_i = 0;
        end
    end

// provide the signal to the channels (4 state machines)

  assign ch1_n_i = !ch1_i;
  assign ch2_n_i = !ch2_i;
  assign ch3_n_i = !ch3_i;
  assign ch4_n_i = !ch4_i;

// combinatorial ch1temp
  always@(currentState1 or gateCh1 or gateEnPulse)
    begin
      case (currentState1)
        idle :
          begin
            ch1temp = 0;
            if (gateCh1 == 1 && gateEnPulse == 1)
              nextState1 = active;
            else
              nextState1 = idle;
          end
        active :
          begin
            ch1temp = 1;
            if (gateCh1 == 0 && gateEnPulse == 1)
              nextState1 = idle;
            else
              nextState1 = active;
          end
        default : // should never get here
          begin
            ch1temp = 0;
```

```
            nextState1 = idle;
          end
      endcase
    end

// combinatorial ch2temp
  always@(currentState2 or gateCh2 or gateEnPulse)
    begin
      case (currentState2)
        idle :
          begin
            ch2temp = 0;
            if (gateCh2 == 1 && gateEnPulse == 1)
              nextState2 = active;
            else
              nextState2 = idle;
          end
        active :
          begin
            ch2temp = 1;
            if (gateCh2 == 0 && gateEnPulse == 1)
              nextState2 = idle;
            else
              nextState2 = active;
          end
        default : // should never get here
          begin
            ch2temp = 0;
            nextState2 = idle;
          end
      endcase
    end

// combinatorial ch3temp
  always@(currentState3 or gateCh3 or gateEnPulse)
    begin
      case (currentState3)
        idle :
          begin
            ch3temp = 0;
            if (gateCh3 == 1 && gateEnPulse == 1)
              nextState3 = active;
            else
              nextState3 = idle;
          end
        active :
          begin
            ch3temp = 1;
            if (gateCh3 == 0 && gateEnPulse == 1)
              nextState3 = idle;
            else
              nextState3 = active;
          end
        default : // should never get here
          begin
            ch3temp = 0;
            nextState3 = idle;
          end
      endcase
    end

// combinatorial ch4temp
  always@(currentState4 or gateCh4 or gateEnPulse)
    begin
      case (currentState4)
        idle :
          begin
            ch4temp = 0;
            if (gateCh4 == 1 && gateEnPulse == 1)
```

```
            nextState4 = active;
          else
            nextState4 = idle;
        end
      active :
        begin
          ch4temp = 1;
          if (gateCh4 == 0 && gateEnPulse == 1)
            nextState4 = idle;
          else
            nextState4 = active;
        end
      default : // should never get here
        begin
          ch4temp = 0;
          nextState4 = idle;
        end
    endcase
  end

// sequential state
  always@(posedge clk)
    begin
      if (rst_n == 0)
        begin
          currentState1 <= idle;
          currentState2 <= idle;
          currentState3 <= idle;
          currentState4 <= idle;
        end
      else
        begin
          currentState1 <= nextState1;
          currentState2 <= nextState2;
          currentState3 <= nextState3;
          currentState4 <= nextState4;
        end
    end

// conditionally put output channels in hiZ or grounded
// when not 100% duty cycle
  always@(gndOrHiZDutyCycle or ch1temp or ch2temp or ch3temp or ch4temp or
          fout)
    begin
      if (gndOrHiZDutyCycle == 0)
        begin
          ch1_i = ch1temp;
          ch2_i = ch2temp;
          ch3_i = ch3temp;
          ch4_i = ch4temp;
        end
      else
        begin
          ch1_i = ch1temp & fout;
          ch2_i = ch2temp & fout;
          ch3_i = ch3temp & fout;
          ch4_i = ch4temp & fout;
        end
    end

// register output to deglitch
  always@(posedge clk)
    begin
      if (rst_n == 0)
        begin
          ch1   <= 0;
          ch2   <= 0;
          ch3   <= 0;
          ch4   <= 0;
```

```
            ch1_n <= 1;
            ch2_n <= 1;
            ch3_n <= 1;
            ch4_n <= 1;
            polA  <= 0;
            polB  <= 1;
          end
        else
          begin
            ch1   <= ch1_i;
            ch2   <= ch2_i;
            ch3   <= ch3_i;
            ch4   <= ch4_i;
            ch1_n <= ch1_n_i;
            ch2_n <= ch2_n_i;
            ch3_n <= ch3_n_i;
            ch4_n <= ch4_n_i;
            polA  <= polA_i;
            polB  <= polB_i;
          end
      end

endmodule
```

A.8. Module register.v

```
module register(rst_n,clk,uartDout,rd,serialShift,highCount,
                  lowCount,gndOrHiZDutyCycle,ch1StopCnt,ch1StartCnt,ch2StopCnt,
                  ch2StartCnt,ch3StopCnt,ch3StartCnt,ch4StopCnt,ch4StartCnt,
                  i2cDataOut,regDataOut,regLoad//,errChk
                  );

    `include "paramFile.v"

    input rst_n;
    input clk;
    input [7:0] uartDout;
    input rd;
    input serialShift;
    output [11:0] highCount;
    output [11:0] lowCount;
    output gndOrHiZDutyCycle;
    output [7:0] ch1StopCnt;
    output [7:0] ch1StartCnt;
    output [7:0] ch2StopCnt;
    output [7:0] ch2StartCnt;
    output [7:0] ch3StopCnt;
    output [7:0] ch3StartCnt;
    output [7:0] ch4StopCnt;
    output [7:0] ch4StartCnt;
    input i2cDataOut;
    output regDataOut;
    input regLoad;
  //  output errChk;

    reg [`registerBits - 1:0] shiftReg;

    wire [7:0] lRegVal;
    wire [7:0] sRegVal;
//
  always@(negedge rst_n or posedge clk)
    begin
      if (rst_n == 0 )
        begin
//          shiftReg <= 0; // uncomment for regular operation

// the below set of initialization values is for testing, but may be
// alternatively used as a set of default reset parameters, if desired

          shiftReg[`registerBits - 1 :`registerBits -  8]  <= 8'd 31;//200; // lRegVal
8 bits    1100_1000
          shiftReg[`registerBits - 9 :`registerBits - 16]  <= 8'd 31;//150; // sRegVal
8 bits    1001_0110
          shiftReg[`registerBits - 17:`registerBits - 28]  <= 12'd 328;//100; //
highCount         12 bits  0000_0110_0100
          shiftReg[`registerBits - 29:`registerBits - 40]  <= 12'd 328;//50; // lowCount
12 bits   0000_0011_0010
          shiftReg[`registerBits - 41]                     <= 1'd 0;//1; //
gndOrHiZDutyCycle 1 bit       1
          shiftReg[`registerBits - 42:`registerBits - 49]  <= 8'd 1;//5;  //  ch1StartCnt
8 bits     0000_0101
          shiftReg[`registerBits - 50:`registerBits - 57]  <= 8'd 2;//15; //  ch1StopCnt
8 bits     0000_1111
          shiftReg[`registerBits - 58:`registerBits - 65]  <= 8'd 3;//10; //  ch2StartCnt
8 bits     0000_1010
          shiftReg[`registerBits - 66:`registerBits - 73]  <= 8'd 4;//20; //  ch2StopCnt
8 bits     0001_0100
          shiftReg[`registerBits - 74:`registerBits - 81]  <= 8'd 5;//15;   //
ch3StartCnt      8 bits   0000_1111
          shiftReg[`registerBits - 82:`registerBits - 89]  <= 8'd 6;//25;   //
ch3StopCnt       8 bits   0001_1001
```

```
          shiftReg[`registerBits - 90:`registerBits - 97]  <= 8'd 7;//20;   //
ch4StartCnt      8 bits   0001_0100
          shiftReg[`registerBits - 98:`registerBits - 105] <= 8'd 8;//30;   //
ch4StopCnt       8 bits   0001_1110

        end
      else if (rd == 1) // then load upper 8 bits of shift register
        shiftReg[`registerBits - 1:`registerBits - 8] <= uartDout;
      else if (serialShift == 1 && regLoad == 1)
      // then rotate shift register left by one bit (MSB
      // rotates to LSB) and load bottom bit
        shiftReg <= {shiftReg[`registerBits - 1 - 1:0],i2cDataOut};
      else if (serialShift == 1) // then rotate shift register left by one bit (MSB
                                 // rotates to LSB
        shiftReg <= {shiftReg[`registerBits - 1 - 1:0],shiftReg[`registerBits - 1]};
      else
        shiftReg <= shiftReg; // unnecessary line
    end

assign regDataOut  = shiftReg[`registerBits - 1];

assign lRegVal           = shiftReg[`registerBits - 1 :`registerBits -  8];// 8 bits
assign sRegVal           = shiftReg[`registerBits - 9 :`registerBits - 16];// 8 bits
assign highCount         = shiftReg[`registerBits - 17:`registerBits - 28];// 12 bits
assign lowCount          = shiftReg[`registerBits - 29:`registerBits - 40];// 12 bits
assign gndOrHiZDutyCycle = shiftReg[`registerBits - 41];              // 1 bit
assign ch1StartCnt       = shiftReg[`registerBits - 42:`registerBits - 49];// 8 bits
assign ch1StopCnt        = shiftReg[`registerBits - 50:`registerBits - 57];// 8 bits
assign ch2StartCnt       = shiftReg[`registerBits - 58:`registerBits - 65];// 8 bits
assign ch2StopCnt        = shiftReg[`registerBits - 66:`registerBits - 73];// 8 bits
assign ch3StartCnt       = shiftReg[`registerBits - 74:`registerBits - 81];// 8 bits
assign ch3StopCnt        = shiftReg[`registerBits - 82:`registerBits - 89];// 8 bits
assign ch4StartCnt       = shiftReg[`registerBits - 90:`registerBits - 97];// 8 bits
assign ch4StopCnt        = shiftReg[`registerBits - 98:`registerBits - 105];// 8 bits

//assign errChk      = i2cDataOut ^ regDataOut; // (XOR to check if bits are
                                               // same, if not, then this is
                                               // an error)

endmodule
```

A.9. Module resetBlock.v

```
module resetBlock(rst_n,clk);

    `include "paramFile.v"

    output rst_n;
    input clk;

    reg [3:0] startCount = 4'b 1010; // assume flops are either all zero or all
                                     // one in the real device at power up

    reg [3:0] counter = 4'b 0000; // assigned value for simulation only

    reg rst_n = 1; // assigned value for simulation only

    always@(posedge clk)
      begin
        if(counter == startCount)
          begin
            counter <= counter;
            rst_n   <= 1;
          end
        else
          begin
            counter <= counter + 1;
            rst_n   <= !rst_n; // gives multiple resets until the final count
          end
      end

endmodule
```

A.10. Module sysControl.v

```
   // messages
   // note, nibble commands are coded in Manchester format: 0 = 01, 1 = 10
   //                          Man. binary   nibble hex    dec    ASCII
   `define uartError           8'b 0101_0101 // 0000 : 0x55 : 0d85    U
   `define crcError            8'b 0101_0110 // 0001 : 0x56 : 0d86    V 0100_0010
   `define i2cError            8'b 0101_1001 // 0010 : 0x59 : 0d89    Y
   `define pwmCompleteUART     8'b 0101_1010 // 0011 : 0x5a : 0d90    Z
   `define vRegError           8'b 0110_0101 // 0100 : 0x65 : 0d101   e
   `define sysContErr          8'b 1001_1001 // 1010 : 0x66 : 0d102   f
   `define incompleteUartErr   8'b 1001_1010 // 1011 : 0x9a : 0d154   U with .. on top
   `define manError            8'b 1010_0101 // 1100 : 0xa5 : 0d165   N with .. on top
0100_1000
   `define demodError          8'b 1010_0110 // 1101 : 0xa6 : 0d166   superscript "a" or
backwards "c"
   `define pwmCompleteSprom    8'b 1010_1001 // 1110 : 0xa9 : 0d169   _ or +



module sysControl(rst_n,clk,crcPass,serialShift,
                  dataReady,uartDin,uartEn,enFreq,i2cAddr,
                  i2cComplete,i2cRd,i2cWr,i2cEn,sRegOn,lRegOn_n,
                  lRegErr_n,enChSw,enCrc,uartErr,regLoad,rd,i2cErr,
                  //errChk,
                  txComplete,
                 // readActive,
                  oneHzClk,chSwComplete,wr,manErr,demodErr
//                  ,currentState,nextState
                  );

  `include "paramFile.v"

  input  rst_n;
  input  clk;
  input  crcPass; // level, when CRC is completed, this should be 0, sysControl
                  // checks the timing and value of this signal
  output serialShift; // shifts serial data to register.v and i2c.v, 1 bit per
                      // clk cycle
  input  dataReady; // strobe when data has been received by the uart
  output [7:0] uartDin; // data to be sent through the uart, registered version of
"message"
  output uartEn; // enables uart.v
  output enFreq; // enables freqGenerator.v
  output [15:0] i2cAddr; // address and control word to send out on i2c
  input  i2cComplete; // 0 if processing an i2c transfer, 1 otherwise
  output i2cRd; // strobe to start an i2c read operation
  output i2cWr; // strobe to start an i2c write operation
  output i2cEn; // enables i2c.v
  output sRegOn; // 1 = switching regulator ON, 0 = OFF
  output lRegOn_n; // 1 = linear regulator OFF, 0 = ON
  input  lRegErr_n; // 0 = error signal active on linear regulator (voltage dropout)
  output enChSw; // enable channel switcher
  output enCrc; // enable crc engine
  input  uartErr; // strobe, stop bit error detected by uart
  output regLoad; // loads the i2cDataOut signal to the MSB of the shift register
                  // with no shift, need to use serial shift with this
  output rd; // loads the 8 MSB's of the shift register with uartDout (uart
             // receive data)
  input  i2cErr; // strobes when there is an failure in acknowledgement in i2c
                 // communications
  input  txComplete; // level, 0 when uart is transmitting data
  input  oneHzClk;
  input  chSwComplete; // level, 1 if chSw has completed stimulation of all channels, 0
otherwise
  output wr; // strobe, command to send uartDin out through the uart
  input manErr;
  input demodErr;
```

```
// outputs
reg serialShift;
reg [7:0] uartDin;
reg uartEn;
reg enFreq;
reg [15:0] i2cAddr;
reg i2cRd;
reg i2cWr;
reg i2cEn;
reg sRegOn,sRegOn_int;
reg lRegOn_n,lRegOn_n_int;
reg enChSw;
reg enCrc;
reg regLoad;
reg rd;
reg wr;

// Signals
  reg [4:0] currentState,nextState;
  reg [6:0] secondsCount; // [`secondsCountBits-1:0] secondsCount;
  reg oneHzClk_r;
  reg shiftCommand;
  reg [7:0] shiftValue;
  reg [7:0] shiftCount; // can count to 256
  reg serialShiftActive;
  reg [4:0] byteCount;
  reg [7:0] message;
  reg getMessage;
  reg [6:0] i2cSlaveAddr,i2cSlaveAddr_r;
  reg i2cR_Wn,i2cR_Wn_r;
  reg [7:0] i2cControlByte,i2cControlByte_r;
  reg i2cAddrLoadCommand;
  reg clearByteCount;
  reg countI2cbyte;
  wire oneHzPulse;
  reg [7:0] uartDin_temp;
  reg rxUartData,rxUartData_r;
  reg pwmFromSprom,pwmFromSprom_r;


//detect if a uart transfer has been successful
  always@(posedge clk)
    begin
      if (rst_n == 0)
        pwmFromSprom_r <= 0;
      else if (pwmFromSprom == 1)
        pwmFromSprom_r <= 1;
      else
        pwmFromSprom_r <= pwmFromSprom_r;
    end

//detect if a uart transfer has been successful
  always@(posedge clk)
    begin
      if (rst_n == 0)
        begin
          rxUartData_r <= 0;
        end
      else
        begin
          rxUartData_r <= rxUartData;
        end
    end

// remove possible glitches
  always@(posedge clk)
    begin
      if (rst_n == 0)
        begin
```

```
          lRegOn_n = 1;//off
          sRegOn = 0;  //off
        end
      else
        begin
          lRegOn_n = lRegOn_n_int;
          sRegOn = sRegOn_int;
        end
    end

// uart receive byte counter register
  always@(posedge clk)
    begin
      if (rst_n == 0)
        begin
          byteCount = 5'b 0_0000;
        end
      else if (dataReady == 1 || countI2cbyte == 1)
        begin
          byteCount = byteCount + 1;
        end
      else if (clearByteCount == 1)
        byteCount = 5'b 0_0000;
    end

// i2c address and control byte register
  always@(posedge clk)
    begin
      if (rst_n == 0)
        begin
          i2cSlaveAddr_r   = 7'b 000_0000; //XXX_XXXX;
          i2cR_Wn_r        = 1'b 0; //X;
          i2cControlByte_r = 8'b 0000_0000; //XXXX_XXXX;
        end
      else if (i2cAddrLoadCommand == 1)
        begin
          i2cSlaveAddr_r   = i2cSlaveAddr;
          i2cR_Wn_r        = i2cR_Wn;
          i2cControlByte_r = i2cControlByte;
        end
    end

always@(i2cSlaveAddr_r or i2cR_Wn_r or i2cControlByte_r)
  begin
    i2cAddr[15:9] = i2cSlaveAddr_r;
    i2cAddr[8]    = i2cR_Wn_r;
    i2cAddr[7:0]  = i2cControlByte_r;
  end

// message register
  always@(posedge clk)
    begin
      if (rst_n == 0)
        uartDin_temp <= 8'b 0000_0000;//XXXX_XXXX; // don't care at start up, only when a
value
                                     // gets written will it be sent (so save gates)
      else if (getMessage == 1)
        uartDin_temp <= message;
    end

  always@(uartDin_temp)
    begin
      uartDin[7] = uartDin_temp[0]; //reverse order because uart sends LSB first
      uartDin[6] = uartDin_temp[1]; //reverse order because uart sends LSB first
      uartDin[5] = uartDin_temp[2]; //reverse order because uart sends LSB first
      uartDin[4] = uartDin_temp[3]; //reverse order because uart sends LSB first
      uartDin[3] = uartDin_temp[4]; //reverse order because uart sends LSB first
      uartDin[2] = uartDin_temp[5]; //reverse order because uart sends LSB first
      uartDin[1] = uartDin_temp[6]; //reverse order because uart sends LSB first
```

```
      uartDin[0] = uartDin_temp[7]; //reverse order because uart sends LSB first
    end

// serialShift control (note, can shorten this code by having default values)
  always@(posedge clk)
    begin
      if (rst_n == 0)
        begin
          shiftCount        <= 0;
          serialShift       <= 0;
          serialShiftActive <= 0;
        end
      else if (shiftCommand == 1) // strobe from main FSM to load the shiftValue
        begin
          shiftCount        <= shiftValue;
          serialShift       <= 0;
          serialShiftActive <= 1;
        end
      else if (shiftCount == 0)
        begin
          serialShift       <= 0;
          serialShiftActive <= 0;
        end
      else
        begin
          serialShift       <= 1;
          shiftCount        <= shiftCount - 1;
          serialShiftActive <= 1;
        end
    end

// seconds counter (note this works in simulation and rtl versions too)
assign oneHzPulse = ((oneHzClk_r ^ oneHzClk) & oneHzClk);

  always@(posedge clk)
    begin
      if (rst_n == 0)
        begin
          oneHzClk_r <= 0;
        end
      else
        begin
          oneHzClk_r <= oneHzClk;
        end
    end
  always@(posedge clk)
    begin
      if (rst_n == 0)
        begin
          secondsCount <= 0;
        end
      else if (oneHzPulse == 1)
        begin
          secondsCount <= secondsCount + 1;
        end
      else
        begin
          secondsCount <= secondsCount; // don't need this line because this is clocked
process
        end
    end

// Main State Machine for sysControl

  // FSM1 States
  parameter idleS0                  =  0,
            waitForDataS1           =  1,
            getUartDataS2           =  2,
            loadToI2cS3             =  3,
```

```
            returnMessageS4          =  4,
            computeCrcS5             =  5,
            checkCRCS6               =  6,
            loadFromI2cS7            =  7,
            waitI2cRegWriteS8        =  8,
            regWriteS9               =  9,
            enablePwmEngineS10       = 10,
            writeI2cToLVRegS11       = 11,
            waitI2cToLVRegS12        = 12,
            writeI2cToSVRegS13       = 13,
            waitI2cToSVRegS14        = 14,
            repeatMessageS15         = 15,
            shiftI2cReg2SpromS16     = 16,
            waitI2cReg2SpromWriteS17 = 17,
            regWriteS18              = 18
            ;

  // combinatorial
  always@(currentState      or
          dataReady         or
          secondsCount      or
          uartErr           or
          i2cComplete       or
          byteCount         or
          serialShiftActive or
          lRegErr_n         or
          chSwComplete      or
          i2cErr            or
          txComplete        or
          oneHzPulse        or
          crcPass           or
          rxUartData_r
         )
    begin
    // default statements

      // default output signals
//      uartDin             = 8'b 0000_0000;
      uartEn              = 0;
      enFreq              = 0;
      i2cSlaveAddr        = 7'b 000_0000;//XXX_XXXX;
      i2cR_Wn             = 0;
      i2cControlByte      = 8'b 0000_0000;//XXXX_XXXX;
      i2cRd               = 0;
      i2cWr               = 0;
      i2cEn               = 0;
      sRegOn_int          = 1;//on
      lRegOn_n_int        = 0;//on
      enChSw              = 0;
      enCrc               = 0;
      regLoad             = 0;
      rd                  = 0;
      wr                  = 0;
      // default internal state machine signals
      nextState           = 1;
      getMessage          = 0;
      message             = 0;
      shiftCommand        = 0;
      i2cAddrLoadCommand  = 0;
      clearByteCount      = 0;
      countI2cbyte        = 0;
      shiftValue          = 8'd 0;
      rxUartData          = 0;
      pwmFromSprom        = 0;

      case (currentState)
// S0
        idleS0 :
          begin
```

```
            if (secondsCount == `secondsCountBits'd 30)//30)
            begin
                sRegOn_int = 0;  //off
                lRegOn_n_int = 1;//off
                if (demodErr == 1)
                  begin
                    getMessage = 1;
                    message    = `demodError; // code for demodulator error
                    nextState  = returnMessageS4;
                  end
            // then 30 seconds have elapsed and power should be stable, can count
            // to more seconds, if required
              nextState = waitForDataS1;
            end
            else
              begin
                nextState   = idleS0;
              end
          end
// S1
        waitForDataS1 :
          begin
            uartEn = 1; // enable the uart to check for messages
              if (uartErr == 1)
                begin
                  getMessage = 1;
                  message    = `uartError; // code for uart error
                  nextState  = returnMessageS4;
                end
              else if (manErr == 1)
                begin
                  getMessage = 1;
                  message    = `manError; // code for uart error
                  nextState  = returnMessageS4;
                end
              else if (dataReady == 1)
                nextState = getUartDataS2;
              else if (secondsCount == `secondsCountBits'd 120) //120) // wait 90 seconds
(120-30) for messages
                begin
                  if (rxUartData_r == 1)
                    begin
                      getMessage = 1;
                      message    = `incompleteUartErr; // code for uart error where a
byte was received but uart timed out before the entire packet was received
                      nextState  = returnMessageS4;
                    end
                  else
                    begin
                    // timed out, load the registers from the sprom
                      i2cEn     = 1;
                      pwmFromSprom = 1;
                      nextState = loadFromI2cS7;
                    end
                end
              else
                nextState = waitForDataS1;
          end

// S2 receive uart
        getUartDataS2 :
          begin
            rxUartData = 1;
            uartEn = 1;
            rd = 1; // load the 8 MSB in shift register
            shiftCommand = 1; // use this to kick off a counter to control the
                              // serialShift signal
            shiftValue = 8'd 8; // shift 8 bits
            if (byteCount == `numRegisterBytes)
```

```
            begin
            // this checks if all the expected bytes have been received
            // if true then, go compute the CRC
              clearByteCount = 1; //clears the byte counter
              nextState = computeCrcS5;
            end
          else
          // wait for the next byte or timeout
            nextState = waitForDataS1;
        end

// S3 load sprom with register data, via i2c
      loadToI2cS3 :
        begin
          i2cEn     = 1;
          if (serialShiftActive == 1)// then wait for i2c register to be
                                     // read before starting another transfer
            begin
              nextState = loadToI2cS3; // go back and wait for i2c engine to be
available
            end
          else
            begin
              i2cAddrLoadCommand = 1;
              i2cSlaveAddr       = 7'b 101_0000;//101_0XXX; // lower bytes are don't
care per sprom datasheet, so save some logic
              i2cR_Wn            = 0; // this will be a read operation
              i2cWr              = 1;
              i2cControlByte     = {4'b 0000,byteCount}; //{4'b XXXX,byteCount}; //
this is the sprom memory address, upper nibble is don't care, so save some logic
              nextState          = waitI2cReg2SpromWriteS17;
            end
        end

// S4 send error/status message back to belt
      returnMessageS4 :
            begin
              uartEn = 1; // enable the uart
              sRegOn_int = 0;  //off
              lRegOn_n_int = 1;//off
              wr          = 1;
              nextState   = repeatMessageS15;
            end

// S5 compute the crc
      computeCrcS5 :
        begin
          if (serialShiftActive == 1)// then wait for register to finish shifting
            begin
              nextState = computeCrcS5; // go back and wait for i2c engine to be
available
            end
          else
            begin
              // shift all the register bits through crc engine
              shiftCommand = 1; // kicks off shift engine
              shiftValue   = 8'd `numRegisterBytes*8; //number of reg bits, 18*8
              // shiftValue   = 8'd 14*8; // use this to determine the CRC value
              enCrc        = 1;
              nextState    = checkCRCS6;
            end
        end

// S6 check if the crc passed
      checkCRCS6 :
        begin
          enCrc = 1;
          if (serialShiftActive == 1)
            // then still shifting through CRC
```

```
            nextState = checkCRCS6;
          else // finished shifting, now check result
            begin
              if (crcPass == 1)
                begin
                  clearByteCount = 1; // ensure byte counter is zeroed
//nextState = regWriteS18;
                  nextState = shiftI2cReg2SpromS16;
                end
              else
                begin
                  getMessage = 1; // command to store message
                  message    = `crcError; // code for crc error
                  nextState  = returnMessageS4;
                end
            end
        end

// S7 load registers with sprom data, via i2c
      loadFromI2cS7 :
        begin
          i2cEn     = 1;
          if (serialShiftActive == 1)// then wait for i2c register to be
                                     // read before starting another transfer
            begin
              regLoad   = 1; // need to keep loading the register
              nextState = loadFromI2cS7; // go back and wait for i2c engine to be
available
            end
          else if (byteCount == 5'b 1_0000) // completed 16 bytes in sprom
            begin
              clearByteCount = 1; // does actually does nothing, does need to at this
point
              shiftCommand = 1;
              shiftValue   = 8'd 24; // shift 16 bits to clear the CRC register bits
and get the data to the top of the register
                                     // then shift 8 bits more to load the i2c register
              nextState = writeI2cToLVRegS11;
            end
          else
            begin
              i2cAddrLoadCommand = 1;
              i2cSlaveAddr       = 7'b 101_0000;//101_0XXX; // lower bytes are don't
care per sprom datasheet, so save some logic
              i2cR_Wn            = 1; // this will be a read operation
              i2cRd              = 1;
              i2cControlByte     = {4'b 0000,byteCount};//{4'b XXXX,byteCount}; // this
is the sprom memory address, upper nibble is don't care, so save some logic
              nextState          = waitI2cRegWriteS8;
            end
        end

// S8
      waitI2cRegWriteS8 :
        begin
          i2cEn = 1;
          if (i2cErr == 1) // of course, the error condition gets priority
            begin
              getMessage = 1;
              message    = `i2cError; // code for i2c error
              nextState  = returnMessageS4;
            end
          else if (i2cComplete == 1) // then write the received i2c byte to the
register
            nextState = regWriteS9;
          else
            nextState = waitI2cRegWriteS8; // go back and wait for i2c transfer to
complete
        end
```

```
// S9 for sprom to reg loading
        regWriteS9 :
          begin
            i2cEn        = 1;
            countI2cbyte = 1; // increment the sprom memory address (and byte count)
            shiftCommand = 1;
            shiftValue   = 8'd 8; // shift 8 bits
            regLoad      = 1;
            nextState = loadFromI2cS7;
          end

// S10
        enablePwmEngineS10 :
          begin
            sRegOn_int = 0;
            lRegOn_n_int = 0;
            if (i2cErr == 1) // of course, the error condition gets priority
              begin
                getMessage = 1;
                message    = `i2cError; // code for i2c error
                nextState  = returnMessageS4;
              end
            else if (serialShiftActive == 1)// then wait for i2c register to be
                                            // read before starting another transfer
              begin
                i2cEn     = 1;
                nextState = enablePwmEngineS10; // go back and wait for i2c engine to be
available
              end
            else
              begin
                enFreq = 1;
                enChSw = 1;
                if (lRegErr_n == 0)
                  begin
                    getMessage = 1;
                    message    = `vRegError; // code for voltage regulation error
                    nextState  = returnMessageS4;
                  end
                else if (chSwComplete == 1)
                  begin
                    getMessage = 1;
                    if (pwmFromSprom_r == 0)
                      message    = `pwmCompleteUART; // code for stimulation complete,
pattern from uart
                    else
                      message    = `pwmCompleteSprom; // code for stimulation complete,
pattern from sprom
                    nextState  = returnMessageS4;
                  end

                else // wait here until PWM engine completes
                  nextState = enablePwmEngineS10;
              end
          end

// S11
        writeI2cToLVRegS11 :
          begin
            i2cEn = 1;
            if (i2cErr == 1) // of course, the error condition gets priority
              begin
                getMessage = 1;
                message    = `i2cError; // code for i2c error
                nextState  = returnMessageS4;
              end
            else if (serialShiftActive == 1)// then wait for i2c register to be
                                            // read before starting another transfer
```

```
            begin
              nextState = writeI2cToLVRegS11; // go back and wait for i2c engine to be
available
            end
          else
            begin
              i2cAddrLoadCommand = 1;
              i2cSlaveAddr       = 7'b 010_1100; // linear regulator's digital pot
address
              i2cR_Wn            = 0; // this will be a write operation
              i2cWr              = 1;
              i2cControlByte     = {8'b 0000_0000};//{8'b X00X_XXXX}; // control byte
for write, mostly is don't care, so save some logic
              nextState          = waitI2cToLVRegS12;
            end
        end

// S12
      waitI2cToLVRegS12 :
        begin
          i2cEn = 1;
          if (i2cErr == 1) // of course, the error condition gets priority
            begin
              getMessage = 1;
              message    = `i2cError; // code for i2c error
              nextState  = returnMessageS4;
            end
          else if (i2cComplete == 1) // then write the received i2c byte to the
register
            begin
              shiftCommand = 1;
              shiftValue   = 8'd 8; // shift 8 bits to load i2c
              nextState    = writeI2cToSVRegS13;
            end
          else
            nextState = waitI2cToLVRegS12; // go back and wait for i2c transfer to
complete
        end

// S13
      writeI2cToSVRegS13 :
        begin
          i2cEn = 1;
          if (i2cErr == 1) // of course, the error condition gets priority
            begin
              getMessage = 1;
              message    = `i2cError; // code for i2c error
              nextState  = returnMessageS4;
            end
          else if (serialShiftActive == 1)// then wait for i2c register to be
                                          // read before starting another transfer
            begin
              nextState = writeI2cToSVRegS13; // go back and wait for i2c engine to be
available
            end
          else
            begin
              i2cAddrLoadCommand = 1;
              i2cSlaveAddr       = 7'b 010_1101; // switching regulator's digital pot
address
              i2cR_Wn            = 0; // this will be a write operation
              i2cWr              = 1;
              i2cControlByte     = {8'b 0000_0000};//{8'b X00X_XXXX}; // control byte
for write, mostly is don't care, so save some logic
              nextState          = waitI2cToSVRegS14;
            end
        end

// S14
```

```
        waitI2cToSVRegS14 :
          begin
            i2cEn = 1;
            if (i2cErr == 1) // of course, the error condition gets priority
              begin
                getMessage = 1;
                message    = `i2cError; // code for i2c error
                nextState  = returnMessageS4;
              end
            else if (i2cComplete == 1) // then write the received i2c byte to the
register
              begin
                shiftCommand = 1;
                shiftValue   = 8'd 128; // shift 128 bits to rotate register back to
starting point
                nextState = enablePwmEngineS10;
              end
            else
              nextState = waitI2cToSVRegS14; // go back and wait for i2c transfer to
complete
          end

// S15
        repeatMessageS15 :
          begin
            uartEn = 1; // enable the uart
            sRegOn_int = 0;  //off
            lRegOn_n_int = 1;//off
            if (txComplete == 1 && oneHzPulse == 1)
            // wait for message to be sent and for 1 Hz interval to elapse
              nextState = returnMessageS4; // repeat the message
            else // wait for message to be sent and 1 second repeat interval
              nextState = repeatMessageS15;
          end


// S16 shift 8 bits from register to the i2c register
        shiftI2cReg2SpromS16 :
          begin
            i2cEn          = 1;
            shiftCommand   = 1;
            shiftValue     = 8'd 8; // shift 8 bits to load i2c
            nextState      = loadToI2cS3;
          end

// S17
        waitI2cReg2SpromWriteS17 :
          begin
            i2cEn = 1;
            if (i2cErr == 1) // of course, the error condition gets priority
              begin
                getMessage = 1;
                message    = `i2cError; // code for i2c error
                nextState  = returnMessageS4;
              end
            else if (i2cComplete == 1) // then write the received i2c byte to the
register
              begin
                if (byteCount == 5'b 0_1111) // 16 bytes in sprom
                  nextState = regWriteS18;
                else // keep going to next byte to be transferred
                  begin
                    countI2cbyte   = 1; // increment the sprom memory address (and byte
count)
                    nextState = shiftI2cReg2SpromS16;
                  end
              end
            else
```

```
            nextState = waitI2cReg2SpromWriteS17; // go back and wait for i2c
transfer to complete
          end

// S18 for reg to sprom loading
         regWriteS18 :
           begin
             i2cEn        = 1;
             countI2cbyte = 1; // increment the sprom memory address (and byte count)
             shiftCommand = 1;
             shiftValue   = 8'd 8; // shift 8 bits
             regLoad      = 1;
             if (byteCount == 5'b 0_1111) // 16 bytes in sprom
               begin
                 clearByteCount = 1; // does actually does nothing, does need to at this
point
                 shiftValue   = 8'd 24; // shift 16 bits to clear the CRC register bits
and get the data to the top of the register
//                 shiftValue   = 8'd 8;    // then shift 8 bits more to load the i2c
register
                 nextState = writeI2cToLVRegS11; // finished so go write to voltage
regulators
               end
             else
               nextState = loadFromI2cS7;
           end

   // default case
         default :
         begin
           getMessage = 1; // command to store message
           message    =`sysContErr; // code for sysControl error
           nextState  = returnMessageS4;
         end
       endcase
     end

   // sequential state
   always@(posedge clk)
     begin
       if (rst_n == 0)
         currentState <= idleS0;
       else
         currentState <= nextState;
     end

 endmodule
```

A.11. Module timer.v

```
`timescale 1 ns/1 ps

module timer(rst_n,clk,oneHzClk,ACKclk);

    `include "paramFile.v"

    input rst_n;
    input clk;
    output oneHzClk;
    output ACKclk;

    reg timer_7_r;

  reg[`timerBits-1:0] timer; //free running counter for clock scaling

  always@(posedge clk) // clock scaling
    begin
      if(rst_n == 0)
        timer <= 0;
      else
        timer <= timer + 1;
    end

  always@(posedge clk)
    begin
      if(rst_n == 0)
        timer_7_r <= 0;
      else
        timer_7_r <= timer[7];
    end

  assign ACKclk = ((timer_7_r ^ timer[7]) & timer[7]);//generate a pulse

//`else
  assign oneHzClk = timer[14]; // this is the correct 1 Hz clock
//`endif

endmodule
```

A.12. Testbench top_tb.v

```
// TEST 0 begin: test getting uart data into registers, no errors
// Meant to pass with code 0x5a

// TEST 1 begin: test getting uart data into registers, shows CRC error
// Meant to fail with code 0x56

// TEST 2 begin: test getting uart data into registers, shows Manchester error
// Meant to fail with code 0x55

// TEST 3 begin: test getting data into registers from sprom, pwm engine
// activating and completing and sending uart message back to indicate compelete
// Meant to pass with code 0x5a

// TEST 4 begin:  test lRegErr_n signal (fails), give uart message
// Meant to fail with code 0x65

// TEST 5 begin:  test i2c error
// Meant to fail with code 0x59

`timescale 1 ns/1 ps

module top_tb();

parameter clkper = 1000000000/32768; // clkper in ns
parameter clkperdiv4 = clkper/8; // clkper in ns
parameter uartBitPeriod = 1000000000/1200; // uart bit period in ns


// tb signals
  reg [3:0] TEST_NUMBER;

  reg [10:0] rxByte;
  reg [7:0] rxByteTemp;

  reg [10:0] rxByteTempCRC_0;
  reg [10:0] rxByteTempCRC_1;
  reg [10:0] rxByteTempCRC_2;
  reg [10:0] rxByteTempCRC_3;
  reg [10:0] rxByteTempCRC_4;
  reg [10:0] rxByteTempCRC_5;
  reg [10:0] rxByteTempCRC_6;
  reg [10:0] rxByteTempCRC_7;

// Inputs
//    reg rst_n;
    reg clk;
    reg rxD;
    reg lRegErr_n;


// Outputs
    wire SCL;
    wire txD;
    wire sRegOn;
    wire lRegOn_n;
    wire ch1,ch2,ch3,ch4;
    wire ch1_n,ch2_n,ch3_n,ch4_n;
    wire polA,polB;

//    wire [4:0] currentState,nextState;
//    wire i2cErr,i2cComplete;
//    wire [143:0] shiftReg;
//    wire regLoad,rd,serialShift;
//  wire shiftReg0;
//  wire shiftReg1;
//  wire shiftReg2;
//  wire shiftReg3;
```

```
// Bidirs
    wire SDA;
    reg  SDA_in;
    reg  SDA_inTemp;
    reg  TRI_STATE;


assign SDA =(TRI_STATE == 1) ? SDA_in : 1'b z;
// TRI_STATE = 1, DUT SDA is an input
// TRI_STATE = 0, DUT SDA is an output

// Instantiate the UUT
    top top (
     //   .rst_n(rst_n),
        .clk(clk),
        .SDA(SDA),
        .SCL(SCL),
        .rxD(rxD),
        .txD(txD),
        .sRegOn(sRegOn),
        .lRegOn_n(lRegOn_n),
        .lRegErr_n(lRegErr_n)
        ,.ch1(ch1),.ch2(ch2),.ch3(ch3),.ch4(ch4)
        ,.ch1_n(ch1_n),.ch2_n(ch2_n),.ch3_n(ch3_n),.ch4_n(ch4_n)
        ,.polA(polA),.polB(polB)
//        ,.currentState(currentState)
//        ,.nextState(nextState)
//        ,.i2cErr(i2cErr)
//        ,.i2cComplete(i2cComplete)
//        ,.regLoad(regLoad),.rd(rd),.serialShift(serialShift)
//        ,.shiftReg(shiftReg)
//           ,.shiftReg0(shiftReg0)
//           ,.shiftReg1(shiftReg1)
//           ,.shiftReg2(shiftReg2)
//           ,.shiftReg3(shiftReg3)
        );

// Initialize Inputs

        initial begin
            // rst_n = 1;
            clk = 0;
            rxD = 1; // idle state is 1
            lRegErr_n = 1; // 1 is no error
            TRI_STATE = 0; // DUT SDA is an output
            TEST_NUMBER = 0;
        end

always # (clkper/2) clk = !clk;

  initial
    begin : uartRxByte
      integer n;
      integer m;
      integer p;


//note these are Manchester encoded so there are 8 instead of 4 CRC bytes
      rxByteTempCRC_0 = 11'b 0_10011001_01; // 1010  0xA
      rxByteTempCRC_1 = 11'b 0_10011010_01; // 1011  0xB
      rxByteTempCRC_2 = 11'b 0_10011001_01; // 1010  0xA
      rxByteTempCRC_3 = 11'b 0_01100110_01; // 0101  0x5
      rxByteTempCRC_4 = 11'b 0_10101001_01; // 1110  0xE
      rxByteTempCRC_5 = 11'b 0_10011010_01; // 1011  0xB
      rxByteTempCRC_6 = 11'b 0_10010110_01; // 1001  0x9
      rxByteTempCRC_7 = 11'b 0_01101010_01; // 0111  0x7
//
```

```
// TEST 0 begin: test getting uart data into registers, no errors
// Meant to pass with code 0x5a
    //  TEST_NUMBER = TEST_NUMBER + 1;
      repeat (2) #clkper;
      // rst_n = 0;
      repeat (2) #clkper;
      // rst_n = 1; // complete reset sequence
      repeat (5) #clkper;
      #clkperdiv4; // some offset to view inputs better

      repeat (2*4*16*31) #clkper; // delay until DUT can accept uart input



      rxByteTemp = 8'b 1001_1001; //0_1001_1001_01; = 1010 = 0xA, !1010 = 0101 = 0x5

      for (m = 0 ; m < 28; m = m + 1) // transfer 28 bytes (14 bytes in Manchester)
        begin
          rxByte = {1'b 0,rxByteTemp,2'b 01}; // Start bit, data byte, Stop bit, idle
          for (p = 0 ; p < 8 ; p = p + 1)
            rxByteTemp[p] = !rxByteTemp[p];
          for (n = 0 ; n < 11 ; n = n + 1)
            begin
              rxD = rxByte[10 - n]; #uartBitPeriod;
            end

         repeat (2) #clkper;  // #uartBitPeriod;
        end


// CRC start, these are CRC bits
      for (m = 0 ; m < 8; m = m + 1) // transfer 4 bytes (8 bytes in Manchester)
        begin
          case (m)
            0 : rxByte = rxByteTempCRC_0; // Start bit, data byte, Stop bit, idle
            1 : rxByte = rxByteTempCRC_1; // Start bit, data byte, Stop bit, idle
            2 : rxByte = rxByteTempCRC_2; // Start bit, data byte, Stop bit, idle
            3 : rxByte = rxByteTempCRC_3; // Start bit, data byte, Stop bit, idle
            4 : rxByte = rxByteTempCRC_4; // Start bit, data byte, Stop bit, idle
            5 : rxByte = rxByteTempCRC_5; // Start bit, data byte, Stop bit, idle
            6 : rxByte = rxByteTempCRC_6; // Start bit, data byte, Stop bit, idle
            7 : rxByte = rxByteTempCRC_7; // Start bit, data byte, Stop bit, idle
            default : rxByte = rxByteTempCRC_0;
          endcase

          for (n = 0 ; n < 11 ; n = n + 1)
            begin
              rxD = rxByte[10 - n]; #uartBitPeriod;
            end

          repeat (2) #clkper;  // #uartBitPeriod;
        end

// i2c write to sprom
/*
      repeat (44 + 16 + 96) #clkper;

      for (m = 0 ; m < 16; m = m + 1) // write 16 bytes to sprom
        begin
          TRI_STATE = 1; // DUT SDA is an input
          SDA_in    = 0;
          repeat (4) #clkper;
          TRI_STATE = 0; // SDA is an output

          repeat (24) #clkper;
```

```
          TRI_STATE = 1; // DUT SDA is an input
          SDA_in    = 0;
          repeat (4) #clkper;
          TRI_STATE = 0; // SDA is an output

          repeat (25) #clkper;

          TRI_STATE = 1; // DUT SDA is an input
          SDA_in    = 0;
          repeat (4) #clkper;
          TRI_STATE = 0; // SDA is an output

          repeat (32+8 + 411) #clkper;
        end
*/
// i2c write to VRegs

      repeat (131 + 10) #clkper;
      repeat (16) #clkper;

      for (m = 0 ; m < 2; m = m + 1) // write 2 bytes VReg's
        begin
          TRI_STATE = 1; // DUT SDA is an input
          SDA_in    = 0;
          repeat (4) #clkper;
          TRI_STATE = 0; // SDA is an output

          repeat (25) #clkper;

          TRI_STATE = 1; // DUT SDA is an input
          SDA_in    = 0;
          repeat (4) #clkper;
          TRI_STATE = 0; // SDA is an output

          repeat (24) #clkper;

          TRI_STATE = 1; // DUT SDA is an input
          SDA_in    = 0;
          repeat (4) #clkper;
          TRI_STATE = 0; // SDA is an output

          repeat (32+7) #clkper;
        end

        repeat (15000) #clkper;

// TEST 0 end: getting uart data into registers, no errors

// TEST 1 begin: test getting uart data into registers, shows CRC error
// Meant to fail with code 0x56

      TEST_NUMBER = TEST_NUMBER + 1;
      repeat (2) #clkper;
      // rst_n = 0;
      repeat (2) #clkper;
      // rst_n = 1; // complete reset sequence
      repeat (5) #clkper;
      #clkperdiv4; // some offset to view inputs better

      repeat (2*4*16*31) #clkper; // delay until DUT can accept uart input


      rxByteTemp = 8'b 1001_1001; //0_1001_1001_01;

      for (m = 0 ; m < 36; m = m + 1) // transfer 36 bytes (18 bytes in Manchester)
        begin
          rxByte = {1'b 0,rxByteTemp,2'b 01}; // Start bit, data byte, Stop bit, idle
          for (p = 0 ; p < 8 ; p = p + 1)
```

```
            rxByteTemp[p] = !rxByteTemp[p];
          for (n = 0 ; n < 11 ; n = n + 1)
            begin
              rxD = rxByte[10 - n]; #uartBitPeriod;
            end

          repeat (2) #clkper;  // #uartBitPeriod;
        end

        repeat (1500) #clkper;

// TEST 1 end: getting uart data into registers, shows CRC error


// TEST 2 begin: test getting uart data into registers, shows Manchester error
// Meant to fail with code 0x55

      TEST_NUMBER = TEST_NUMBER + 1;
      repeat (2) #clkper;
      // rst_n = 0;
      repeat (2) #clkper;
      // rst_n = 1; // complete reset sequence
      repeat (5) #clkper;
      #clkperdiv4; // some offset to view inputs better

      repeat (2*4*16*31) #clkper; // delay until DUT can accept uart input

      rxByteTemp = 8'b 1011_1001; // Manchester error

      for (m = 0 ; m < 3; m = m + 1) // transfer 36 bytes (18 bytes in Manchester)
        begin
          rxByte = {1'b 0,rxByteTemp,2'b 01}; // Start bit, data byte, Stop bit, idle
          for (p = 0 ; p < 8 ; p = p + 1)
            rxByteTemp[p] = !rxByteTemp[p];
          for (n = 0 ; n < 11 ; n = n + 1)
            begin
              rxD = rxByte[10 - n]; #uartBitPeriod;
            end

          #uartBitPeriod;
        end

        repeat (1500) #clkper;

// TEST 2 end: getting uart data into registers, shows Manchester error

// TEST 3 begin: test getting data into registers from sprom, pwm engine
// activating and completing and sending uart message back to indicate compelete
// Meant to pass with code 0x5a

      TEST_NUMBER = TEST_NUMBER + 1;
      repeat (200) #clkper;
      // rst_n = 0;
      repeat (2) #clkper;
      // rst_n = 1; // complete reset sequence
      repeat (5) #clkper;
      #clkperdiv4; // some offset to view inputs better

      repeat (4*7*16*31 - 9) #clkper; // delay until DUT can accept uart input


      SDA_inTemp   = 1; // this is the data

      repeat (1936 - 16*31 - 59-16 + 32 + 2) #clkper;// wait until implant can accept
input
/*
      for (m = 0 ; m < 16; m = m + 1) // read 16 bytes from sprom
        begin
```

```
          repeat (59-16) #clkper;

          TRI_STATE = 1; // DUT SDA is an input
          SDA_in    = 0;
          repeat (4) #clkper;
          TRI_STATE = 0; // SDA is an output

          repeat (26) #clkper;

          TRI_STATE = 1; // DUT SDA is an input
          SDA_in    = 0;
          repeat (4) #clkper;
          TRI_STATE = 0; // SDA is an output

          repeat (26) #clkper;

          TRI_STATE = 1; // DUT SDA is an input
          SDA_in    = 0; // this is the data
          repeat (6) #clkper;
          SDA_in    = !SDA_inTemp; // this is the data
          SDA_inTemp    = !SDA_inTemp;
          repeat (4) #clkper;
          SDA_in    = !SDA_inTemp; // this is the data
          SDA_inTemp    = !SDA_inTemp;
          repeat (6) #clkper;
          SDA_in    = !SDA_inTemp; // this is the data
          SDA_inTemp    = !SDA_inTemp;
          repeat (4+12-6-6) #clkper;
          TRI_STATE = 0; // SDA is an output

        //  repeat (59-16) #clkper;

        end
*/
      repeat (44 + 16 + 10 - 28 + 10) #clkper;

      for (m = 0 ; m < 2; m = m + 1) // write 2 bytes VReg's
        begin
          TRI_STATE = 1; // DUT SDA is an input
          SDA_in    = 0;
          repeat (4) #clkper;
          TRI_STATE = 0; // SDA is an output

          repeat (24) #clkper;

          TRI_STATE = 1; // DUT SDA is an input
          SDA_in    = 0;
          repeat (4) #clkper;
          TRI_STATE = 0; // SDA is an output

          repeat (24) #clkper;

          TRI_STATE = 1; // DUT SDA is an input
          SDA_in    = 0;
          repeat (4) #clkper;
          TRI_STATE = 0; // SDA is an output

          repeat (32+8) #clkper;

        end

       repeat (30000) #clkper;

// TEST 3 end: complete testing of sprom read and pwm engine activation

// TEST 4 begin:  test lRegErr_n signal (fails), give uart message
// Meant to fail with code 0x65

      TEST_NUMBER = TEST_NUMBER + 1;
```

```
      repeat (200) #clkper;
      // rst_n = 0;
      repeat (2) #clkper;
      // rst_n = 1; // complete reset sequence
      repeat (5) #clkper;
   //   #clkperdiv4; // some offset to view inputs better

      repeat (4*7*16*31 - 9) #clkper; // delay until DUT can accept uart input


      SDA_inTemp    = 1; // this is the data

      repeat (1936 - 16*31 - 59-16 + 32) #clkper;// wait until implant can accept input
/*
      for (m = 0 ; m < 16; m = m + 1) // read 16 bytes from sprom
        begin

          repeat (59-16) #clkper;

          TRI_STATE = 1; // DUT SDA is an input
          SDA_in    = 0;
          repeat (4) #clkper;
          TRI_STATE = 0; // SDA is an output

          repeat (26) #clkper;

          TRI_STATE = 1; // DUT SDA is an input
          SDA_in    = 0;
          repeat (4) #clkper;
          TRI_STATE = 0; // SDA is an output

          repeat (26) #clkper;

          TRI_STATE = 1; // DUT SDA is an input
          SDA_in    = 0; // this is the data
          repeat (6) #clkper;
          SDA_in    = !SDA_inTemp; // this is the data
          SDA_inTemp    = !SDA_inTemp;
          repeat (4) #clkper;
          SDA_in    = !SDA_inTemp; // this is the data
          SDA_inTemp    = !SDA_inTemp;
          repeat (6) #clkper;
          SDA_in    = !SDA_inTemp; // this is the data
          SDA_inTemp    = !SDA_inTemp;
          repeat (4+12-6-6) #clkper;
          TRI_STATE = 0; // SDA is an output

        //  repeat (59-16) #clkper;

        end
*/
       repeat (44 + 16 + 10 - 25 + 10) #clkper;

       for (m = 0 ; m < 2; m = m + 1) // write 2 bytes VReg's
         begin
           TRI_STATE = 1; // DUT SDA is an input
           SDA_in    = 0;
           repeat (4) #clkper;
           TRI_STATE = 0; // SDA is an output

           repeat (24) #clkper;

           TRI_STATE = 1; // DUT SDA is an input
           SDA_in    = 0;
           repeat (4) #clkper;
           TRI_STATE = 0; // SDA is an output

           repeat (24) #clkper;
```

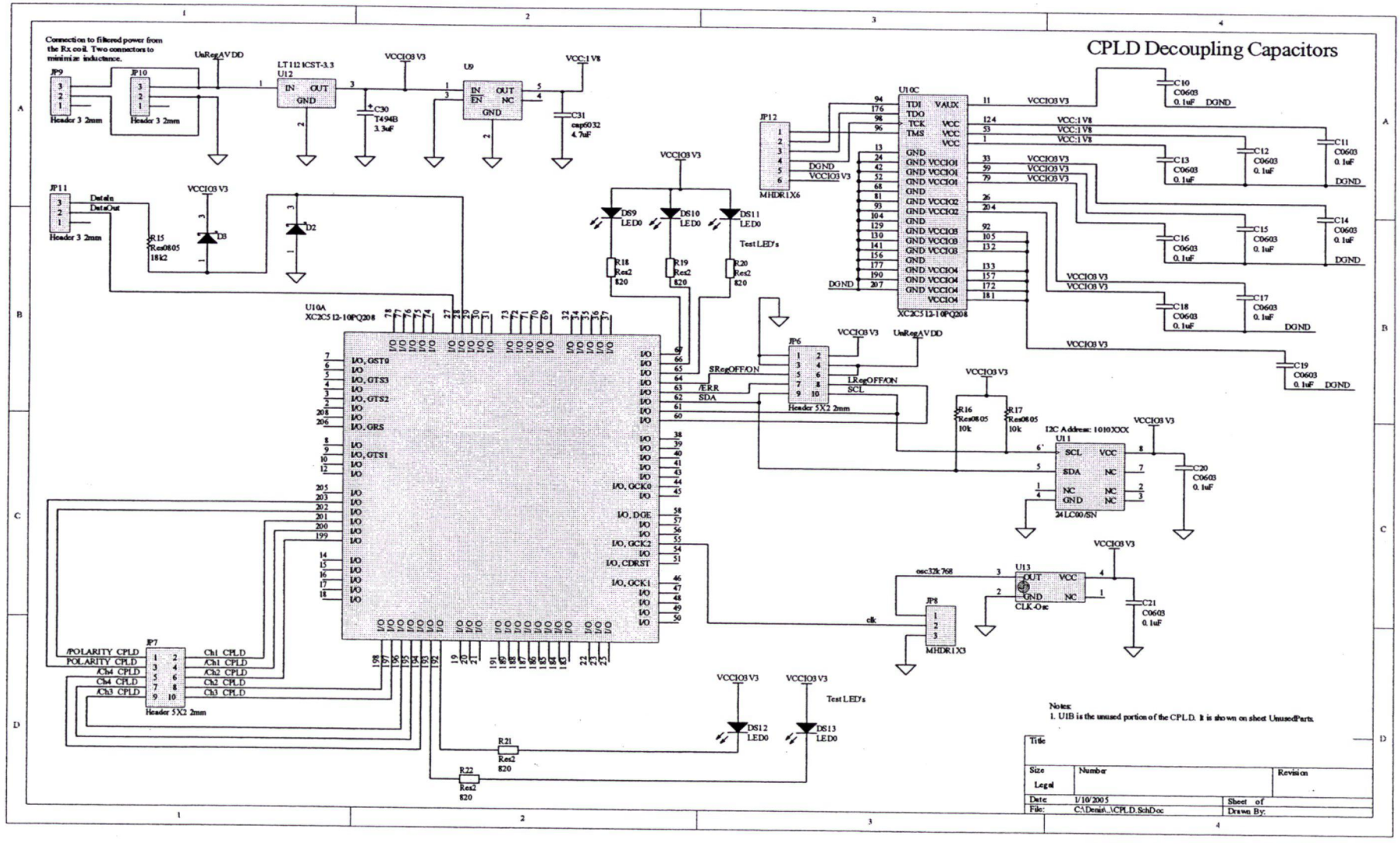
```
          TRI_STATE = 1; // DUT SDA is an input
          SDA_in    = 0;
          repeat (4) #clkper;
          TRI_STATE = 0; // SDA is an output

          repeat (32+8) #clkper;

        end

          repeat (2500) #clkper;//repeat (15000) #clkper;
          lRegErr_n = 0; // 1 is error
          repeat (5000) #clkper;
// TEST 4 end: test lRegErr_n signal (fails)

// TEST 5 begin:  test i2c error
// Meant to fail with code 0x59

      TEST_NUMBER = TEST_NUMBER + 1;
      repeat (200) #clkper;
      // rst_n = 0;
      repeat (2) #clkper;
      // rst_n = 1; // complete reset sequence
      repeat (5) #clkper;
   //   #clkperdiv4; // some offset to view inputs better

      repeat (100) #clkper; // delay until DUT can accept uart input


TRI_STATE = 1; // DUT SDA is an input
SDA_in    = 1; // this is the data

      repeat (1936 - 16*31 - 59-16 + 32 + 2) #clkper;// wait until implant can accept
input
/*
      for (m = 0 ; m < 16; m = m + 1) // read 16 bytes from sprom
        begin

          repeat (59-16) #clkper;

          TRI_STATE = 1; // DUT SDA is an input
          SDA_in    = 0;
          repeat (4) #clkper;
          TRI_STATE = 0; // SDA is an output

          repeat (26) #clkper;

          TRI_STATE = 1; // DUT SDA is an input
          SDA_in    = 0;
          repeat (4) #clkper;
          TRI_STATE = 0; // SDA is an output

          repeat (26) #clkper;

          TRI_STATE = 1; // DUT SDA is an input
          SDA_in    = 0; // this is the data
          repeat (6) #clkper;
          SDA_in    = !SDA_inTemp; // this is the data
          SDA_inTemp    = !SDA_inTemp;
          repeat (4) #clkper;
          SDA_in    = !SDA_inTemp; // this is the data
          SDA_inTemp    = !SDA_inTemp;
          repeat (6) #clkper;
          SDA_in    = !SDA_inTemp; // this is the data
          SDA_inTemp    = !SDA_inTemp;
          repeat (4+12-6-6) #clkper;
          TRI_STATE = 0; // SDA is an output
        end
*/
```

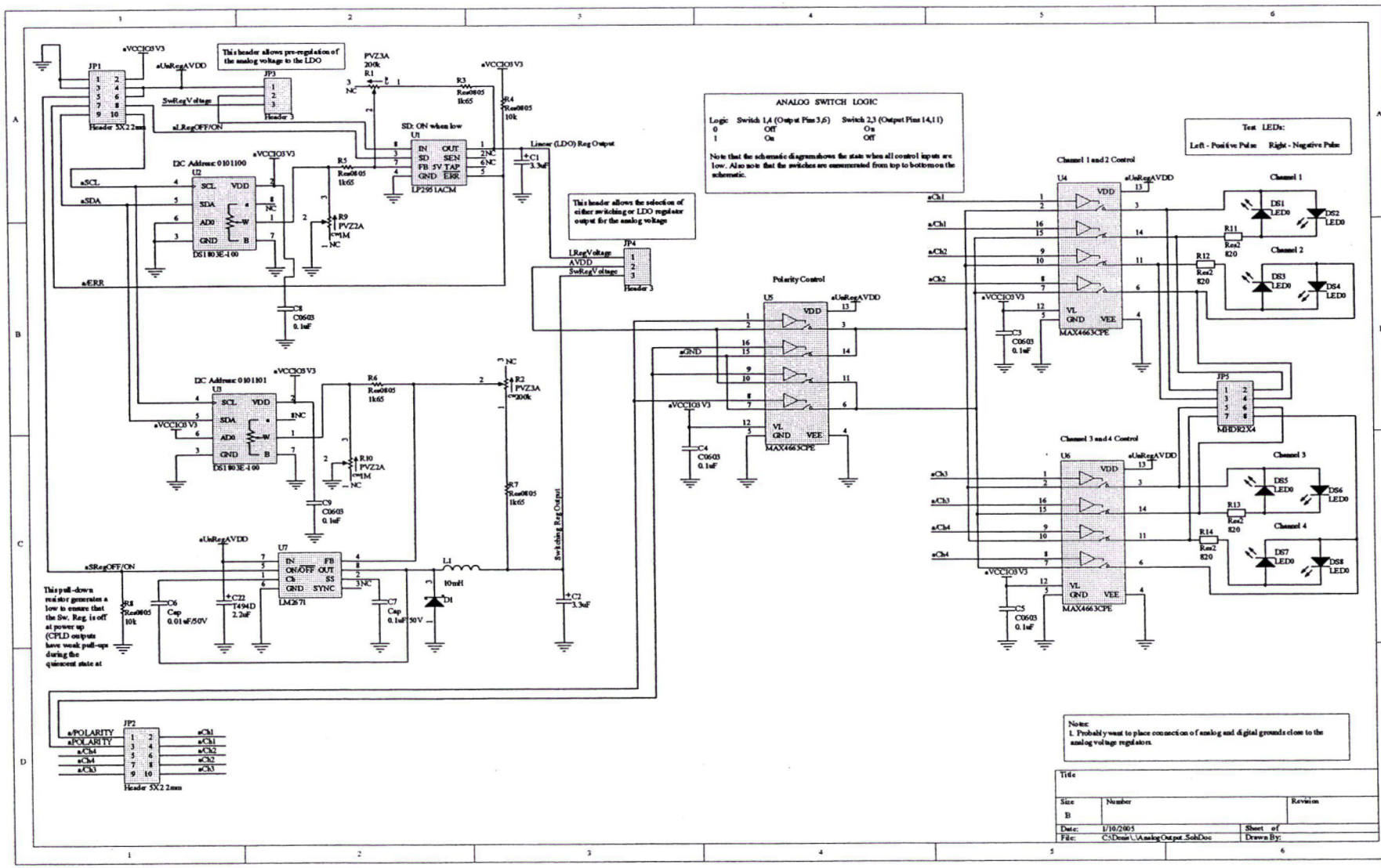
```
// TEST 5 end: test i2c error

      repeat (5000) #clkper;

    end
endmodule
```

A.13. Xilinx User Constraint File top.ucf

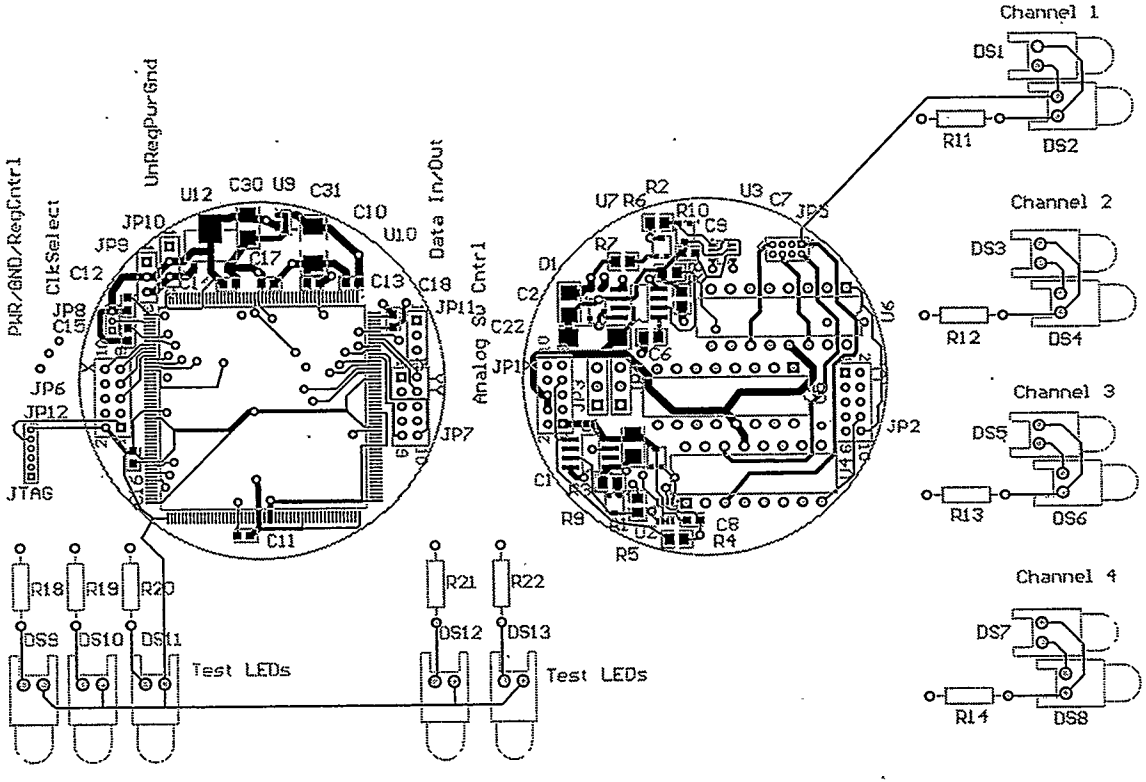
```
#PACE: Start of Constraints generated by PACE

#PACE: Start of PACE I/O Pin Assignments
NET "ch1"  LOC = "p201" | SLEW = SLOW  | IOSTANDARD = LVCMOS33  | KEEPER ;
NET "ch1_n"  LOC = "p200" | SLEW = SLOW  | IOSTANDARD = LVCMOS33  | PULLUP ;
NET "ch2"  LOC = "p198" | SLEW = SLOW  | IOSTANDARD = LVCMOS33  | KEEPER ;
NET "ch2_n"  LOC = "p199" | SLEW = SLOW  | IOSTANDARD = LVCMOS33  | KEEPER ;
NET "ch3"  LOC = "p197" | SLEW = SLOW  | IOSTANDARD = LVCMOS33  | KEEPER ;
NET "ch3_n"  LOC = "p196" | SLEW = SLOW  | IOSTANDARD = LVCMOS33  | KEEPER ;
NET "ch4"  LOC = "p195" | SLEW = SLOW  | IOSTANDARD = LVCMOS33  | KEEPER ;
NET "ch4_n"  LOC = "p194" | SLEW = SLOW  | IOSTANDARD = LVCMOS33  | KEEPER ;
NET "clk"  LOC = "p55" | SCHMITT_TRIGGER  | IOSTANDARD = LVCMOS33  | KEEPER ;
NET "lRegErr_n"  LOC = "p63" | SCHMITT_TRIGGER  | IOSTANDARD = LVCMOS33  | KEEPER ;
NET "lRegOn_n"  LOC = "p60" | SLEW = SLOW  | IOSTANDARD = LVCMOS33  | KEEPER ;
NET "polA"  LOC = "p203" | SLEW = SLOW  | IOSTANDARD = LVCMOS33  | KEEPER ;
NET "polB"  LOC = "p202" | SLEW = SLOW  | IOSTANDARD = LVCMOS33  | KEEPER ;
NET "rxD"  LOC = "p28" | SCHMITT_TRIGGER  | IOSTANDARD = LVCMOS33  | KEEPER ;
NET "SCL"  LOC = "p61" | SLEW = SLOW  | IOSTANDARD = LVCMOS33  | KEEPER ;
NET "SDA"  LOC = "p62" | SLEW = SLOW  | SCHMITT_TRIGGER  | IOSTANDARD = LVCMOS33  |
KEEPER ;
NET "sRegOn"  LOC = "p64" | SLEW = SLOW  | IOSTANDARD = LVCMOS33  | KEEPER ;
NET "txD"  LOC = "p27" | SLEW = SLOW  | IOSTANDARD = LVCMOS33  | KEEPER ;

#PACE: Start of PACE Area Constraints

#PACE: Start of PACE Prohibit Constraints

#PACE: End of Constraints generated by PACE
```

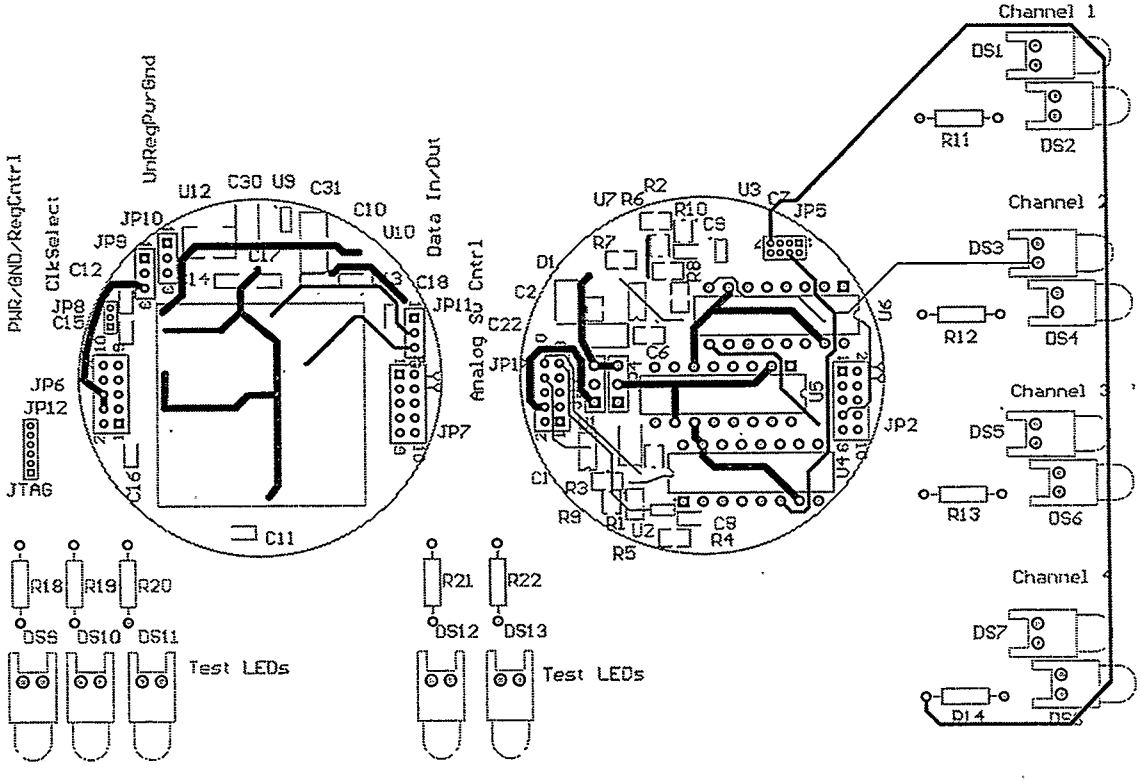
APPENDIX B: SCHEMATIC DIAGRAMS

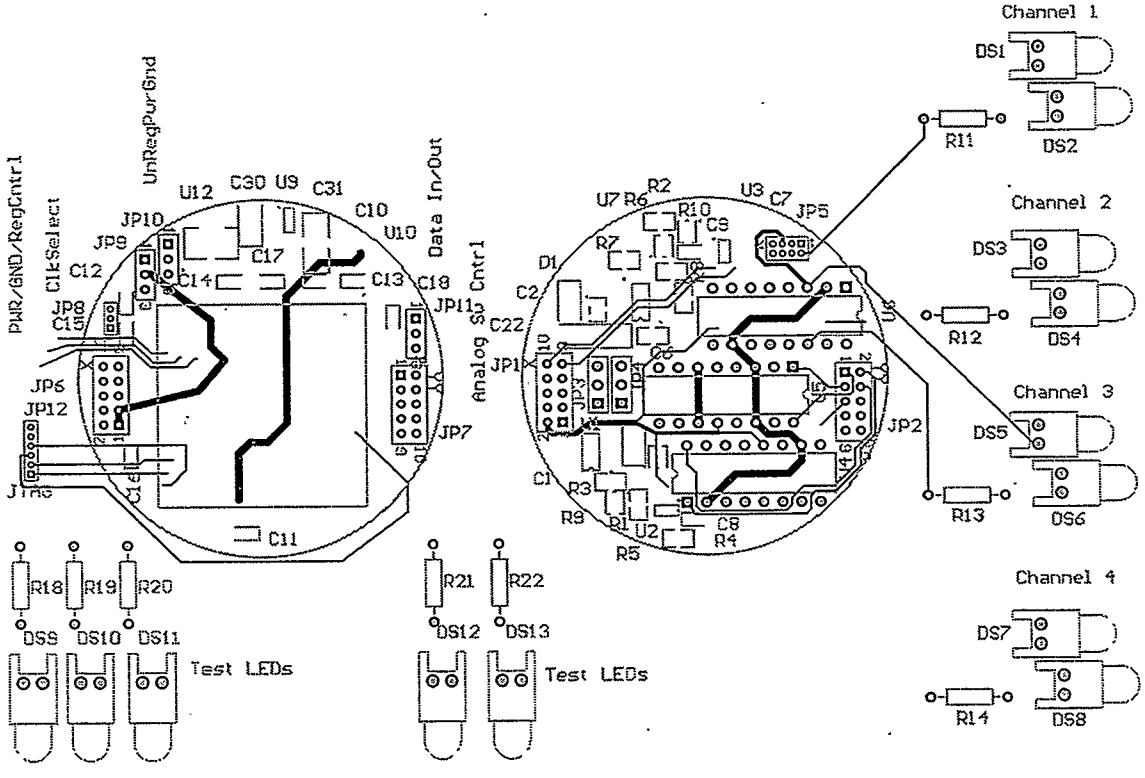

CPLD Decoupling Capacitors
Connection to filtered power from the Rx coil. Two connectors to minimize inductance.
UnRegAVDD
VCCIO3 V3
VCC1 V8
JP9
JP10
Header 3 2mm
U12
LT1121CST-3.3
U9
C30
T494B
3.3uF
C31
cap6032
4.7uF
JP11
DataIn
DataOut
R15
Res0805
18k2
D3
D2
DS9
DS10
DS11
LED0
Test LED's
R18
R19
R20
Res2
820
U10A
XC2C512-10PQ208
I/O, GST0
I/O, GTS3
I/O, GTS2
I/O, GRS
I/O, GTS1
I/O, GCK0
I/O, DGE
I/O, GCK2
I/O, CDRST
I/O, GCK1
JP6
Header 5X2 2mm
SRegOFF/ON
/ERR
SDA
LRegOFF/ON
SCL
R16
R17
Res0805
10k
I2C Address: 1010XXX
U11
24LC00/SN
SCL
SDA
VCC
GND
NC
C20
C0603
0.1uF
U13
CLK-Osc
osc32k768
JP8
MHDR1X3
clk
C21
JP7
Header 5X2 2mm
/POLARITY CPLD
POLARITY CPLD
/Ch4 CPLD
Ch4 CPLD
/Ch3 CPLD
Ch1 CPLD
/Ch1 CPLD
/Ch2 CPLD
Ch2 CPLD
Ch3 CPLD
DS12
DS13
R21
R22
JP12
MHDR1X6
DGND
U10C
XC2C512-10PQ208
C10
C11
C12
C13
C14
C15
C16
C17
C18
C19
Notes:
1. UIB is the unused portion of the CPLD. It is shown on sheet UnusedParts.
Title
Size
Legal
Number
Revision
Date
1/10/2005
Sheet of
File:
C:\Denis\..\CPLD.SchDoc
Drawn By:

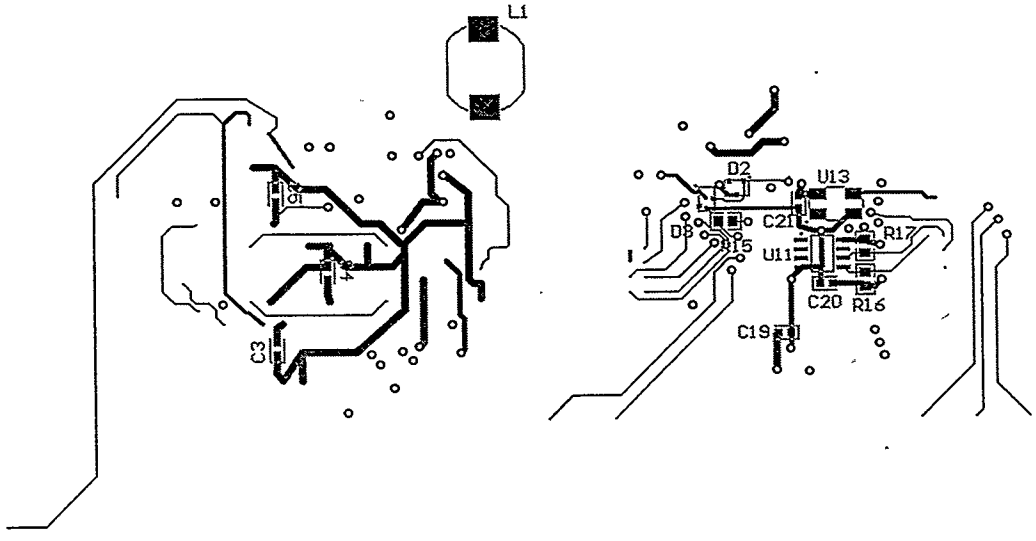

This header allows pre-regulation of the analog voltage to the LDO
ANALOG SWITCH LOGIC
Logic Switch 1,4 (Output Pins 3,6) Switch 2,3 (Output Pins 14,11)
0 Off On
1 On Off
Note that the schematic diagram shows the state when all control inputs are low. Also note that the switches are enumerated from top to bottom on the schematic.
Test LEDs:
Left - Positive Pulse Right - Negative Pulse
This header allows the selection of either switching or LDO regulator output for the analog voltage
Polarity Control
Channel 1 and 2 Control
Channel 3 and 4 Control
Channel 1
Channel 2
Channel 3
Channel 4
Linear (LDO) Reg Output
Switching Reg Output
This pull-down resistor generates a low to ensure that the Sw. Reg. is off at power up (CPLD outputs have weak pull-ups during the quiescent state at
Notes:
1. Probably want to place connection of analog and digital grounds close to the analog voltage regulators.
Title
Size B
Number
Revision
Date: 1/10/2005
Sheet of
File: C:\Denis\...\AnalogOutput_SchDoc
Drawn By:

APPENDIX C: PRINTED CIRCUIT BOARD DIAGRAMS


PWR/GND/RegCntrl
ClkSelect
UnRegPwrGnd
U12
C30
U9
C31
C10
U10
Data In/Out
JP10
JP9
C12
C17
C13
C18
JP11
JP8
C1
JP6
JP12
JP7
JTAG
C11
Analog Sig Cntrl
U7
R6
R2
U3
C7
JP5
R10
C9
R7
D1
C2
C22
JP1
C6
U6
JP3
JP2
C1
R9
C8
R4
R5
Channel 1
DS1
R11
DS2
Channel 2
DS3
R12
DS4
Channel 3
DS5
R13
DS6
Channel 4
DS7
R14
DS8
R18
R19
R20
DS9
DS10
DS11
Test LEDs
R21
R22
DS12
DS13
Test LEDs


UnRegPwr Gnd
PWR/GND/RegCntrl
ClkSelect
Data In/Out
Analog SW Cntrl
JTAG
Test LEDs
Test LEDs
Channel 1
Channel 2
Channel 3
Channel 4


PWR/GND/RegCntrl
ClkSelect
UnRegPwrGnd
Data In/Out
Analog Sw Cntrl
Channel 1
Channel 2
Channel 3
Channel 4
Test LEDs
Test LEDs
DS1
DS2
DS3
DS4
DS5
DS6
DS7
DS8
DS9
DS10
DS11
DS12
DS13
R11
R12
R13
R14
R18
R19
R20
R21
R22
JP1
JP2
JP5
JP6
JP7
JP8
JP9
JP10
JP11
JP12
U12
C30
U9
C31
C10
U10
C12
C13
C14
C15
C17
C18
C11
U7
R6
R2
R10
C9
R7
D1
C2
C22
C1
R3
R9
R1
U2
R5
C8
R4
U3
C7
C6


L1
D2
U13
C21
D3
R15
U11
R17
C20
R16
C19
C3

APPENDIX D: TABLE OF PROGRAMMBLE SETTINGS TO CONTROL NEUROSTIMULATOR FREQUENCY AND DUTY CYCLE

Desired Frequency (Hz)	Desired Duty Cycle (%)	High Count	Low Count	% Error in frequency	% Error in high count	% Error in low count	Sum of absolute values of % errors
250	100	66	0	-0.7	-0.7	0.0	1.4
250	90	59	7	-0.7	0.0	-6.8	7.5
250	80	52	13	0.8	0.8	0.8	2.5
250	70	46	20	-0.7	-0.3	-1.7	2.7
250	60	39	26	0.8	0.8	0.8	2.5
250	50	33	33	-0.7	-0.7	-0.7	2.1
250	40	26	39	0.8	0.8	0.8	2.5
250	30	20	46	-0.7	-1.7	-0.3	2.7
250	20	13	52	0.8	0.8	0.8	2.5
250	10	7	59	-0.7	-6.8	0.0	7.5
240	100	68	0	0.4	0.4	0.0	0.8
240	90	61	7	0.4	0.7	-2.5	3.6
240	80	54	14	0.4	1.1	-2.5	4.1
240	70	48	20	0.4	-0.4	2.3	3.2
240	60	41	27	0.4	-0.1	1.1	1.6
240	50	34	34	0.4	0.4	0.4	1.2
240	40	27	41	0.4	1.1	-0.1	1.6
240	30	20	48	0.4	2.3	-0.4	3.2
240	20	14	54	0.4	-2.5	1.1	4.1
240	10	7	61	0.4	-2.5	0.7	3.6
230	100	71	0	0.3	0.3	0.0	0.7

Desired Frequency (Hz)	Desired Duty Cycle (%)	High Count	Low Count	% Error in frequency	% Error in high count	% Error in low count	Sum of absolute values of % errors
230	90	64	7	0.3	0.2	1.7	2.2
230	80	57	14	0.3	0.0	1.7	2.1
230	70	50	21	0.3	-0.3	1.7	2.3
230	60	43	28	0.3	-0.6	1.7	2.7
230	50	35	36	0.3	1.7	-1.1	3.1
230	40	28	43	0.3	1.7	-0.6	2.7
230	30	21	50	0.3	1.7	-0.3	2.3
230	20	14	57	0.3	1.7	0.0	2.1
230	10	7	64	0.3	1.7	0.2	2.2
220	100	74	0	0.6	0.6	0.0	1.3
220	90	67	7	0.6	0.0	6.0	6.7
220	80	60	15	-0.7	-0.7	-0.7	2.1
220	70	52	22	0.6	0.3	1.5	2.4
220	60	45	30	-0.7	-0.7	-0.7	2.1
220	50	37	37	0.6	0.6	0.6	1.9
220	40	30	45	-0.7	-0.7	-0.7	2.1
220	30	22	52	0.6	1.5	0.3	2.4
220	20	15	60	-0.7	-0.7	-0.7	2.1
220	10	7	67	0.6	6.0	0.0	6.7
210	100	78	0	0.0	0.0	0.0	0.0
210	90	70	8	0.0	0.3	-2.5	2.9
210	80	62	16	0.0	0.7	-2.5	3.2
210	70	55	23	0.0	-0.7	1.7	2.5
210	60	47	31	0.0	-0.4	0.7	1.1

Desired Frequency (Hz)	Desired Duty Cycle (%)	High Count	Low Count	% Error in frequency	% Error in high count	% Error in low count	Sum of absolute values of % errors
210	50	39	39	0.0	0.0	0.0	0.1
210	40	31	47	0.0	0.7	-0.4	1.1
210	30	23	55	0.0	1.7	-0.7	2.5
210	20	16	62	0.0	-2.5	0.7	3.2
210	10	8	70	0.0	-2.5	0.3	2.9
200	100	82	0	-0.1	-0.1	0.0	0.2
200	90	74	8	-0.1	-0.4	2.3	2.8
200	80	66	16	-0.1	-0.7	2.3	3.1
200	70	57	25	-0.1	0.6	-1.7	2.4
200	60	49	33	-0.1	0.3	-0.7	1.1
200	50	41	41	-0.1	-0.1	-0.1	0.3
200	40	33	49	-0.1	-0.7	0.3	1.1
200	30	25	57	-0.1	-1.7	0.6	2.4
200	20	16	66	-0.1	2.3	-0.7	3.1
200	10	8	74	-0.1	2.3	-0.4	2.8
190	100	86	0	0.3	0.3	0.0	0.5
190	90	77	9	0.3	0.8	-4.4	5.4
190	80	69	17	0.3	0.0	1.4	1.7
190	70	60	26	0.3	0.6	-0.5	1.4
190	60	52	34	0.3	-0.5	1.4	2.2
190	50	43	43	0.3	0.3	0.3	0.8
190	40	34	52	0.3	1.4	-0.5	2.2
190	30	26	60	0.3	-0.5	0.6	1.4
190	20	17	69	0.3	1.4	0.0	1.7

Desired Frequency (Hz)	Desired Duty Cycle (%)	High Count	Low Count	% Error in frequency	% Error in high count	% Error in low count	Sum of absolute values of % errors
190	10	9	77	0.3	-4.4	0.8	5.4
180	100	91	0	0.0	0.0	0.0	0.0
180	90	82	9	0.0	-0.1	1.1	1.2
180	80	73	18	0.0	-0.3	1.1	1.4
180	70	64	27	0.0	-0.4	1.1	1.6
180	60	55	36	0.0	-0.7	1.1	1.9
180	50	45	46	0.0	1.1	-1.1	2.2
180	40	36	55	0.0	1.1	-0.7	1.9
180	30	27	64	0.0	1.1	-0.4	1.6
180	20	18	73	0.0	1.1	-0.3	1.4
180	10	9	82	0.0	1.1	-0.1	1.2
170	100	96	0	0.4	0.4	0.0	0.8
170	90	87	10	-0.6	-0.3	-3.8	4.7
170	80	77	19	0.4	0.1	1.4	2.0
170	70	67	29	0.4	0.7	-0.3	1.4
170	60	58	39	-0.6	-0.3	-1.2	2.1
170	50	48	48	0.4	0.4	0.4	1.2
170	40	39	58	-0.6	-1.2	-0.3	2.1
170	30	29	67	0.4	-0.3	0.7	1.4
170	20	19	77	0.4	1.4	0.1	2.0
170	10	10	87	-0.6	-3.8	-0.3	4.7
160	100	102	0	0.4	0.4	0.0	0.8
160	90	92	10	0.4	0.2	2.3	2.9
160	80	82	20	0.4	-0.1	2.3	2.8

Desired Frequency (Hz)	Desired Duty Cycle (%)	High Count	Low Count	% Error in frequency	% Error in high count	% Error in low count	Sum of absolute values of % errors
160	70	72	31	-0.6	-0.4	-0.9	1.9
160	60	61	41	0.4	0.7	-0.1	1.2
160	50	51	51	0.4	0.4	0.4	1.2
160	40	41	61	0.4	-0.1	0.7	1.2
160	30	31	72	-0.6	-0.9	-0.4	1.9
160	20	20	82	0.4	2.3	-0.1	2.8
160	10	10	92	0.4	2.3	0.2	2.9
150	100	109	0	0.2	0.2	0.0	0.4
150	90	98	11	0.2	0.3	-0.7	1.2
150	80	87	22	0.2	0.4	-0.7	1.4
150	70	76	33	0.2	0.6	-0.7	1.5
150	60	65	44	0.2	0.8	-0.7	1.7
150	50	54	55	0.2	1.1	-0.7	2.0
150	40	44	65	0.2	-0.7	0.8	1.7
150	30	33	76	0.2	-0.7	0.6	1.5
150	20	22	87	0.2	-0.7	0.4	1.4
150	10	11	98	0.2	-0.7	0.3	1.2
140	100	117	0	0.0	0.0	0.0	0.0
140	90	105	12	0.0	0.3	-2.5	2.9
140	80	94	23	0.0	-0.4	1.7	2.2
140	70	82	35	0.0	-0.1	0.3	0.4
140	60	70	47	0.0	0.3	-0.4	0.7
140	50	58	59	0.0	0.9	-0.8	1.7
140	40	47	70	0.0	-0.4	0.3	0.7

Desired Frequency (Hz)	Desired Duty Cycle (%)	High Count	Low Count	% Error in frequency	% Error in high count	% Error in low count	Sum of absolute values of % errors
140	30	35	82	0.0	0.3	-0.1	0.4
140	20	23	94	0.0	1.7	-0.4	2.2
140	10	12	105	0.0	-2.5	0.3	2.9
130	100	126	0	0.0	0.0	0.0	0.0
130	90	113	13	0.0	0.4	-3.1	3.6
130	80	101	25	0.0	-0.2	0.8	1.0
130	70	88	38	0.0	0.3	-0.5	0.8
130	60	76	50	0.0	-0.5	0.8	1.3
130	50	63	63	0.0	0.0	0.0	0.1
130	40	50	76	0.0	0.8	-0.5	1.3
130	30	38	88	0.0	-0.5	0.3	0.8
130	20	25	101	0.0	0.8	-0.2	1.0
130	10	13	113	0.0	-3.1	0.4	3.6
120	100	137	0	-0.3	-0.3	0.0	0.7
120	90	123	14	-0.3	-0.1	-2.5	3.0
120	80	109	27	0.4	0.2	1.1	1.7
120	70	96	41	-0.3	-0.4	-0.1	0.9
120	60	82	55	-0.3	-0.1	-0.7	1.1
120	50	68	68	0.4	0.4	0.4	1.2
120	40	55	82	-0.3	-0.7	-0.1	1.1
120	30	41	96	-0.3	-0.1	-0.4	0.9
120	20	27	109	0.4	1.1	0.2	1.7
120	10	14	123	-0.3	-2.5	-0.1	3.0
110	100	149	0	0.0	0.0	0.0	0.1

Desired Frequency (Hz)	Desired Duty Cycle (%)	High Count	Low Count	% Error in frequency	% Error in high count	% Error in low count	Sum of absolute values of % errors
110	90	134	15	0.0	0.0	-0.7	0.8
110	80	119	30	0.0	0.1	-0.7	0.9
110	70	104	45	0.0	0.3	-0.7	1.0
110	60	89	60	0.0	0.4	-0.7	1.2
110	50	74	75	0.0	0.6	-0.7	1.4
110	40	60	89	0.0	-0.7	0.4	1.2
110	30	45	104	0.0	-0.7	0.3	1.0
110	20	30	119	0.0	-0.7	0.1	0.9
110	10	15	134	0.0	-0.7	0.0	0.8
100	100	164	0	-0.1	-0.1	0.0	0.2
100	90	148	16	-0.1	-0.4	2.3	2.8
100	80	131	33	-0.1	0.1	-0.7	0.9
100	70	115	49	-0.1	-0.3	0.3	0.7
100	60	98	66	-0.1	0.3	-0.7	1.1
100	50	82	82	-0.1	-0.1	-0.1	0.3
100	40	66	98	-0.1	-0.7	0.3	1.1
100	30	49	115	-0.1	0.3	-0.3	0.7
100	20	33	131	-0.1	-0.7	0.1	0.9
100	10	16	148	-0.1	2.3	-0.4	2.8
90	100	182	0	0.0	0.0	0.0	0.0
90	90	164	18	0.0	-0.1	1.1	1.2
90	80	146	36	0.0	-0.3	1.1	1.4
90	70	127	55	0.0	0.3	-0.7	1.1
90	60	109	73	0.0	0.2	-0.3	0.5

Desired Frequency (Hz)	Desired Duty Cycle (%)	High Count	Low Count	% Error in frequency	% Error in high count	% Error in low count	Sum of absolute values of % errors
90	50	91	91	0.0	0.0	0.0	0.1
90	40	73	109	0.0	-0.3	0.2	0.5
90	30	55	127	0.0	-0.7	0.3	1.1
90	20	36	146	0.0	1.1	-0.3	1.4
90	10	18	164	0.0	1.1	-0.1	1.2
80	100	205	0	-0.1	-0.1	0.0	0.2
80	90	184	21	-0.1	0.2	-2.5	2.8
80	80	164	41	-0.1	-0.1	-0.1	0.3
80	70	144	61	-0.1	-0.4	0.7	1.3
80	60	123	82	-0.1	-0.1	-0.1	0.3
80	50	102	103	-0.1	0.4	-0.6	1.1
80	40	82	123	-0.1	-0.1	-0.1	0.3
80	30	61	144	-0.1	0.7	-0.4	1.3
80	20	41	164	-0.1	-0.1	-0.1	0.3
80	10	21	184	-0.1	-2.5	0.2	2.8
70	100	234	0	0.0	0.0	0.0	0.0
70	90	211	23	0.0	-0.2	1.7	1.9
70	80	187	47	0.0	0.1	-0.4	0.6
70	70	164	70	0.0	-0.1	0.3	0.4
70	60	140	94	0.0	0.3	-0.4	0.7
70	50	117	117	0.0	0.0	0.0	0.1
70	40	94	140	0.0	-0.4	0.3	0.7
70	30	70	164	0.0	0.3	-0.1	0.4
70	20	47	187	0.0	-0.4	0.1	0.6

Desired Frequency (Hz)	Desired Duty Cycle (%)	High Count	Low Count	% Error in frequency	% Error in high count	% Error in low count	Sum of absolute values of % errors
70	10	23	211	0.0	1.7	-0.2	1.9
60	100	273	0	0.0	0.0	0.0	0.0
60	90	246	27	0.0	-0.1	1.1	1.2
60	80	218	55	0.0	0.2	-0.7	0.9
60	70	191	82	0.0	0.1	-0.1	0.2
60	60	164	109	0.0	-0.1	0.2	0.3
60	50	136	137	0.0	0.4	-0.3	0.8
60	40	109	164	0.0	0.2	-0.1	0.3
60	30	82	191	0.0	-0.1	0.1	0.2
60	20	55	218	0.0	-0.7	0.2	0.9
60	10	27	246	0.0	1.1	-0.1	1.2
50	100	328	0	-0.1	-0.1	0.0	0.2
50	90	295	33	-0.1	0.0	-0.7	0.8
50	80	262	66	-0.1	0.1	-0.7	0.9
50	70	230	98	-0.1	-0.3	0.3	0.7
50	60	197	131	-0.1	-0.2	0.1	0.4
50	50	164	164	-0.1	-0.1	-0.1	0.3
50	40	131	197	-0.1	0.1	-0.2	0.4
50	30	98	230	-0.1	0.3	-0.3	0.7
50	20	66	262	-0.1	-0.7	0.1	0.9
50	10	33	295	-0.1	-0.7	0.0	0.8
40	100	410	0	-0.1	-0.1	0.0	0.2
40	90	369	41	-0.1	-0.1	-0.1	0.3
40	80	328	82	-0.1	-0.1	-0.1	0.3

Desired Frequency (Hz)	Desired Duty Cycle (%)	High Count	Low Count	% Error in frequency	% Error in high count	% Error in low count	Sum of absolute values of % errors
40	70	287	123	-0.1	-0.1	-0.1	0.3
40	60	246	164	-0.1	-0.1	-0.1	0.3
40	50	205	205	-0.1	-0.1	-0.1	0.3
40	40	164	246	-0.1	-0.1	-0.1	0.3
40	30	123	287	-0.1	-0.1	-0.1	0.3
40	20	82	328	-0.1	-0.1	-0.1	0.3
40	10	41	369	-0.1	-0.1	-0.1	0.3
30	100	546	0	0.0	0.0	0.0	0.0
30	90	491	55	0.0	0.1	-0.7	0.8
30	80	437	109	0.0	0.0	0.2	0.3
30	70	382	164	0.0	0.1	-0.1	0.2
30	60	328	218	0.0	-0.1	0.2	0.3
30	50	273	273	0.0	0.0	0.0	0.1
30	40	218	328	0.0	0.2	-0.1	0.3
30	30	164	382	0.0	-0.1	0.1	0.2
30	20	109	437	0.0	0.2	0.0	0.3
30	10	55	491	0.0	-0.7	0.1	0.8
20	100	819	0	0.0	0.0	0.0	0.0
20	90	737	82	0.0	0.0	-0.1	0.2
20	80	655	164	0.0	0.1	-0.1	0.2
20	70	573	246	0.0	0.1	-0.1	0.2
20	60	491	328	0.0	0.1	-0.1	0.2
20	50	409	410	0.0	0.1	-0.1	0.3
20	40	328	491	0.0	-0.1	0.1	0.2

Desired Frequency (Hz)	Desired Duty Cycle (%)	High Count	Low Count	% Error in frequency	% Error in high count	% Error in low count	Sum of absolute values of % errors
20	30	246	573	0.0	-0.1	0.1	0.2
20	20	164	655	0.0	-0.1	0.1	0.2
20	10	82	737	0.0	-0.1	0.0	0.2
10	100	1638	0	0.0	0.0	0.0	0.0
10	90	1474	164	0.0	0.0	-0.1	0.2
10	80	1311	328	0.0	0.0	-0.1	0.2
10	70	1147	491	0.0	0.0	0.1	0.1
10	60	983	655	0.0	0.0	0.1	0.1
10	50	819	819	0.0	0.0	0.0	0.1
10	40	655	983	0.0	0.1	0.0	0.1
10	30	491	1147	0.0	0.1	0.0	0.1
10	20	328	1311	0.0	-0.1	0.0	0.2
10	10	164	1474	0.0	-0.1	0.0	0.2

APPENDIX E: MATLAB CODE TO GENERATE OPTIMUM AND FREQUENCY AND DUTY CYCLE PARAMETERS

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Name: gastroFreqPwmOptPercentDiff.m
%
% Author: Denis Onen
%
% Date: July 7, 2004
%
% Description: A script to find the optimum discretized frequency and
% high/low time (for duty cycle) parameters for a given clock frequency.
% This algorithm uses the percentage difference to decide which set of counts
% to use to generate the optimal set of counts for the given frequency and
% duty cycle.  Another algorithm, which has been discarded, used squared
% error to select the optimal counts.
%
% Usage: This is a script and no inputs are required.  The
% "countDataBasePercentDiff" matrix contains the optimal set of counts.  The
% parameters can be edited to select a different source clock, set of
% frequency, or set of duty cycles.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

clear;
format compact;

% Parameters
clk = 32768; % in Hz, source clock frequency to be used to generate the new frequency
             % and duty cycle
desiredFreq = [10:10:250]; % in  Hz, e.g. 10, 20, 30, ... , 250
desiredDutyCycle = [10:10:100]; % in percent, e.g. 10, 20, 30, ... , 100

% Calculated parameters
period = 1/clk;
desiredFreqSize = max(size(desiredFreq)); % 25
desiredDutyCycleSize = max(size(desiredDutyCycle)); % 10
q = 0; % countDataBasePercentDiff increment
for j = 1:desiredDutyCycleSize % step through the duty cycles
    for i = 1:desiredFreqSize % step through the frequencies
        q = q + 1;
        freq = desiredFreq(i);
        dutyCycle = desiredDutyCycle(j);
        optFreqCount = (clk/freq)/2; % calculate the optimal clk period count to
                                     % obtain the desired frequency

        % calculate the optimal count to obtain the high phase for the
        % correct duty cycle
        optHighPhaseCount = optFreqCount*dutyCycle/100;

        % calculate the optimal count to obtain the low phase for the correct
        % duty cycle
        optLowPhaseCount = optFreqCount*(100-dutyCycle)/100;

        % get the various types of roundings of the counts to search for the
        % optimal counts in terms of frequency and high/low phases of duty
        % cycle
        highPhaseCountTemp(1) = floor(optHighPhaseCount);
        highPhaseCountTemp(2) = ceil(optHighPhaseCount);

        lowPhaseCountTemp(1) = floor(optLowPhaseCount);
        lowPhaseCountTemp(2) = ceil(optLowPhaseCount);

        % iterate and get optimums for this particular frequency and duty
        % cycle
        oldTotaldiffError = 1000000;
        for k = 1:2
```

```
            for m = 1:2
                freqCountTemp = highPhaseCountTemp(k) + lowPhaseCountTemp(m);
                freqTemp = 1/(freqCountTemp*period); % the new frequency
                diffFreqError = abs(100*(optFreqCount - freqCountTemp)/optFreqCount);
                % diffFreqError = ((optFreqCount - freqCountTemp)/optFreqCount)^2;
                diffHighPhaseError = abs(100*(optHighPhaseCount -
highPhaseCountTemp(k))/optHighPhaseCount);
                if dutyCycle == 100 % check for special case when lowPhase is 0
                    diffLowPhaseError = 0;
                else
                    diffLowPhaseError = abs(100*(optLowPhaseCount -
lowPhaseCountTemp(m))/optLowPhaseCount);
                end

                % the objective function, how about measuring % diff
                % instead? how about recording diffError too in the database?
                newTotaldiffError = diffFreqError + diffHighPhaseError +
diffLowPhaseError; % equally weight these errors for lack of a better algorithm
                % Format of count database, 2-D array,
                % desiredFreqSize x desiredDutyCycleSize deep (25 x 10 =
                % 250 deep in our case)
                % (1)freq (2)duty cycle (3)high count (4)low count (5)% error in
                % freq (6)% error in high count (7)% error in low count
                % (8) sum of absolute values of % errors (this is the
                % objective function to be minimized)
                countDataBasePercentDiff(q,1) = freq;
                countDataBasePercentDiff(q,2) = dutyCycle;

                if newTotaldiffError < oldTotaldiffError % check if you have a new
"optimum"
                    oldTotaldiffError = newTotaldiffError;
                    countDataBasePercentDiff(q,3) = highPhaseCountTemp(k);
                    countDataBasePercentDiff(q,4) = lowPhaseCountTemp(m);
                    countDataBasePercentDiff(q,5) = 100*(optFreqCount -
freqCountTemp)/optFreqCount;
                    countDataBasePercentDiff(q,6) = 100*(optHighPhaseCount -
highPhaseCountTemp(k))/optHighPhaseCount;
                    if dutyCycle == 100 % check for special case when lowPhase is 0
                        countDataBasePercentDiff(q,7) = 0;
                    else
                        countDataBasePercentDiff(q,7) = 100*(optLowPhaseCount -
lowPhaseCountTemp(m))/optLowPhaseCount;
                    end
                    countDataBasePercentDiff(q,8) = newTotaldiffError;
                end % end if
            end % end for
        end % end for
    end % end for
end % end for
```

APPENDIX F: TABLE OF PROGRAMMABLE SETTINGS TO CONTROL NEUROSTIMULATOR OUTPUT VOLTAGE

R1 = 44.6 kΩ, R3 = 1 MΩ, Rab = 100 kΩ, Rw = 50 Ω

Vref = 1.235 V

D	Rwb (Ω)	R2 = R3//Rwb (Ω)	Vout (V)
255	99659.375	90627.5	1.843
254	99268.75	90304.4	1.845
253	98878.125	89981.0	1.847
252	98487.5	89657.4	1.849
251	98096.875	89333.5	1.852
250	97706.25	89009.5	1.854
249	97315.625	88685.2	1.856
248	96925	88360.6	1.858
247	96534.375	88035.9	1.861
246	96143.75	87710.9	1.863
245	95753.125	87385.7	1.865
244	95362.5	87060.2	1.868
243	94971.875	86734.5	1.870
242	94581.25	86408.6	1.872
241	94190.625	86082.5	1.875
240	93800	85756.1	1.877
239	93409.375	85429.5	1.880
238	93018.75	85102.6	1.882
237	92628.125	84775.5	1.885
236	92237.5	84448.2	1.887
235	91846.875	84120.7	1.890
234	91456.25	83792.9	1.892
233	91065.625	83464.8	1.895

D	Rwb (Ω)	R2 = R3//Rwb (Ω)	Vout (V)
232	90675	83136.6	1.898
231	90284.375	82808.1	1.900
230	89893.75	82479.4	1.903
229	89503.125	82150.4	1.905
228	89112.5	81821.2	1.908
227	88721.875	81491.8	1.911
226	88331.25	81162.1	1.914
225	87940.625	80832.2	1.916
224	87550	80502.0	1.919
223	87159.375	80171.7	1.922
222	86768.75	79841.0	1.925
221	86378.125	79510.2	1.928
220	85987.5	79179.1	1.931
219	85596.875	78847.8	1.934
218	85206.25	78516.2	1.937
217	84815.625	78184.4	1.940
216	84425	77852.3	1.943
215	84034.375	77520.0	1.946
214	83643.75	77187.5	1.949
213	83253.125	76854.7	1.952
212	82862.5	76521.7	1.955
211	82471.875	76188.5	1.958
210	82081.25	75855.0	1.961
209	81690.625	75521.2	1.964
208	81300	75187.3	1.968
207	80909.375	74853.1	1.971
206	80518.75	74518.6	1.974
205	80128.125	74183.9	1.977

D	Rwb (Ω)	R2 = R3//Rwb (Ω)	Vout (V)
204	79737.5	73849.0	1.981
203	79346.875	73513.8	1.984
202	78956.25	73178.4	1.988
201	78565.625	72842.7	1.991
200	78175	72506.8	1.995
199	77784.375	72170.6	1.998
198	77393.75	71834.2	2.002
197	77003.125	71497.6	2.005
196	76612.5	71160.7	2.009
195	76221.875	70823.6	2.013
194	75831.25	70486.2	2.016
193	75440.625	70148.6	2.020
192	75050	69810.7	2.024
191	74659.375	69472.6	2.028
190	74268.75	69134.2	2.032
189	73878.125	68795.6	2.036
188	73487.5	68456.8	2.040
187	73096.875	68117.7	2.044
186	72706.25	67778.3	2.048
185	72315.625	67438.7	2.052
184	71925	67098.9	2.056
183	71534.375	66758.8	2.060
182	71143.75	66418.5	2.064
181	70753.125	66077.9	2.069
180	70362.5	65737.1	2.073
179	69971.875	65396.0	2.077
178	69581.25	65054.7	2.082
177	69190.625	64713.1	2.086

D	Rwb (Ω)	R2 = R3//Rwb (Ω)	Vout (V)
176	68800	64371.3	2.091
175	68409.375	64029.2	2.095
174	68018.75	63686.9	2.100
173	67628.125	63344.3	2.105
172	67237.5	63001.4	2.109
171	66846.875	62658.4	2.114
170	66456.25	62315.0	2.119
169	66065.625	61971.4	2.124
168	65675	61627.6	2.129
167	65284.375	61283.5	2.134
166	64893.75	60939.2	2.139
165	64503.125	60594.6	2.144
164	64112.5	60249.7	2.149
163	63721.875	59904.6	2.154
162	63331.25	59559.3	2.160
161	62940.625	59213.7	2.165
160	62550	58867.8	2.171
159	62159.375	58521.7	2.176
158	61768.75	58175.3	2.182
157	61378.125	57828.7	2.187
156	60987.5	57481.8	2.193
155	60596.875	57134.7	2.199
154	60206.25	56787.3	2.205
153	59815.625	56439.7	2.211
152	59425	56091.7	2.217
151	59034.375	55743.6	2.223
150	58643.75	55395.2	2.229
149	58253.125	55046.5	2.236

D	Rwb (Ω)	R2 = R3//Rwb (Ω)	Vout (V)
148	57862.5	54697.6	2.242
147	57471.875	54348.4	2.248
146	57081.25	53998.9	2.255
145	56690.625	53649.2	2.262
144	56300	53299.3	2.268
143	55909.375	52949.0	2.275
142	55518.75	52598.5	2.282
141	55128.125	52247.8	2.289
140	54737.5	51896.8	2.296
139	54346.875	51545.5	2.304
138	53956.25	51194.0	2.311
137	53565.625	50842.2	2.318
136	53175	50490.2	2.326
135	52784.375	50137.9	2.334
134	52393.75	49785.3	2.341
133	52003.125	49432.5	2.349
132	51612.5	49079.4	2.357
131	51221.875	48726.0	2.365
130	50831.25	48372.4	2.374
129	50440.625	48018.5	2.382
128	50050	47664.4	2.391
127	49659.375	47310.0	2.399
126	49268.75	46955.3	2.408
125	48878.125	46600.4	2.417
124	48487.5	46245.2	2.426
123	48096.875	45889.7	2.435
122	47706.25	45534.0	2.445
121	47315.625	45178.0	2.454

D	Rwb (Ω)	R2 = R3//Rwb (Ω)	Vout (V)
120	46925	44821.7	2.464
119	46534.375	44465.2	2.474
118	46143.75	44108.4	2.484
117	45753.125	43751.4	2.494
116	45362.5	43394.0	2.504
115	44971.875	43036.4	2.515
114	44581.25	42678.6	2.526
113	44190.625	42320.5	2.537
112	43800	41962.1	2.548
111	43409.375	41603.4	2.559
110	43018.75	41244.5	2.570
109	42628.125	40885.3	2.582
108	42237.5	40525.8	2.594
107	41846.875	40166.1	2.606
106	41456.25	39806.0	2.619
105	41065.625	39445.8	2.631
104	40675	39085.2	2.644
103	40284.375	38724.4	2.657
102	39893.75	38363.3	2.671
101	39503.125	38001.9	2.684
100	39112.5	37640.3	2.698
99	38721.875	37278.4	2.713
98	38331.25	36916.2	2.727
97	37940.625	36553.8	2.742
96	37550	36191.0	2.757
95	37159.375	35828.0	2.772
94	36768.75	35464.8	2.788
93	36378.125	35101.2	2.804

D	Rwb (Ω)	R2 = R3//Rwb (Ω)	Vout (V)
92	35987.5	34737.4	2.821
91	35596.875	34373.3	2.837
90	35206.25	34008.9	2.855
89	34815.625	33644.3	2.872
88	34425	33279.4	2.890
87	34034.375	32914.2	2.908
86	33643.75	32548.7	2.927
85	33253.125	32182.9	2.946
84	32862.5	31816.9	2.966
83	32471.875	31450.6	2.986
82	32081.25	31084.0	3.007
81	31690.625	30717.2	3.028
80	31300	30350.0	3.050
79	30909.375	29982.6	3.072
78	30518.75	29614.9	3.095
77	30128.125	29247.0	3.118
76	29737.5	28878.7	3.142
75	29346.875	28510.2	3.167
74	28956.25	28141.4	3.192
73	28565.625	27772.3	3.218
72	28175	27402.9	3.245
71	27784.375	27033.3	3.273
70	27393.75	26663.3	3.301
69	27003.125	26293.1	3.330
68	26612.5	25922.6	3.360
67	26221.875	25551.9	3.391
66	25831.25	25180.8	3.422
65	25440.625	24809.5	3.455

D	Rwb (Ω)	R2 = R3//Rwb (Ω)	Vout (V)
64	25050	24437.8	3.489
63	24659.375	24065.9	3.524
62	24268.75	23693.7	3.560
61	23878.125	23321.3	3.597
60	23487.5	22948.5	3.635
59	23096.875	22575.5	3.675
58	22706.25	22202.1	3.716
57	22315.625	21828.5	3.758
56	21925	21454.6	3.802
55	21534.375	21080.4	3.848
54	21143.75	20705.9	3.895
53	20753.125	20331.2	3.944
52	20362.5	19956.1	3.995
51	19971.875	19580.8	4.048
50	19581.25	19205.2	4.103
49	19190.625	18829.3	4.160
48	18800	18453.1	4.220
47	18409.375	18076.6	4.282
46	18018.75	17699.8	4.347
45	17628.125	17322.8	4.415
44	17237.5	16945.4	4.485
43	16846.875	16567.8	4.560
42	16456.25	16189.8	4.637
41	16065.625	15811.6	4.719
40	15675	15433.1	4.804
39	15284.375	15054.3	4.894
38	14893.75	14675.2	4.988
37	14503.125	14295.8	5.088

D	Rwb (Ω)	R2 = R3//Rwb (Ω)	Vout (V)
36	14112.5	13916.1	5.193
35	13721.875	13536.1	5.304
34	13331.25	13155.9	5.422
33	12940.625	12775.3	5.547
32	12550	12394.4	5.679
31	12159.375	12013.3	5.820
30	11768.75	11631.9	5.970
29	11378.125	11250.1	6.131
28	10987.5	10868.1	6.303
27	10596.875	10485.8	6.488
26	10206.25	10103.1	6.687
25	9815.625	9720.2	6.902
24	9425	9337.0	7.134
23	9034.375	8953.5	7.387
22	8643.75	8569.7	7.662
21	8253.125	8185.6	7.964
20	7862.5	7801.2	8.296
19	7471.875	7416.5	8.662
18	7081.25	7031.5	9.069
17	6690.625	6646.2	9.523
16	6300	6260.6	10.033
15	5909.375	5874.7	10.611
14	5518.75	5488.5	11.271
13	5128.125	5102.0	12.031
12	4737.5	4715.2	12.917
11	4346.875	4328.1	13.961
10	3956.25	3940.7	15.213
9	3565.625	3553.0	16.738

D	Rwb (Ω)	R2 = R3//Rwb (Ω)	Vout (V)
8	3175	3165.0	18.638
7	2784.375	2776.6	21.072
6	2393.75	2388.0	24.300
5	2003.125	1999.1	28.788
4	1612.5	1609.9	35.449
3	1221.875	1220.4	46.369
2	831.25	830.6	67.553
1	440.625	440.4	126.297
0	50	50.0	1102.910

Note that the output voltage values for digital potentiometer settings 15 to 0 exceed the required specifications and in some cases, e.g. 3 to 0, are absurd, but included for completeness.