AFIT/GE/ENG/99M-15

Graphical User Interface and Microprocessor Control Enhancement of a Pseudorandom Code Generator

THESIS

John M. Kos, B.S.E.E.
Second Lieutenant, USAF

AFIT/GE/ENG/99M-15

The views expressed in this document are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

AFIT/GE/ENG/98M-15

Graphical User Interface and Microprocessor Control Enhancement of a Pseudorandom Code Generator

THESIS

Presented to the Faculty of the Graduate School of Engineering

of the Air Force Institute of Technology

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Electrical Engineering

John M. Kos, B.S.E.E

Second Lieutenant, USAF

March 1999

AFIT/GE/ENG/98M-15

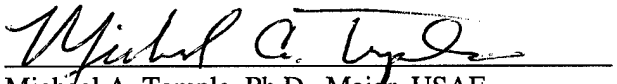
Graphical User Interface and Microprocessor
Control Enhancement of a
Pseudorandom Code Generator

THESIS

John M. Kos, B.S.E.E

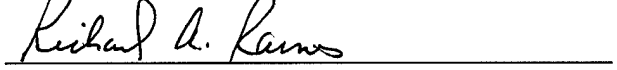
Second Lieutenant, USAF

Approved:



Michael A. Temple, Ph.D., Major, USAF
Committee Chairman

5 Mar 99
Date



Richard A. Raines, Ph.D., Major, USAF
Committee Member

5 Mar 99
Date

Approved for public release; distribution unlimited

ACKNOWLEDGEMENTS

The time has come to put my many thanks into words on a page. I would like to thank my wonderful fiancé, Amanda, for putting up with the many hours I spent sitting at the computer and not with her. A big thanks to my academic and thesis advisor, Major Michael A. Temple, for guiding me to a project that fit my desires—part hardware, part software—and for his help along the way. Thank you, Major Raines, for putting up with me. I thank my sponsor, Mr. James P. Stephens (AFRL/SNRW), for having this project to offer. I am thankful for the good friends I have made in my GE-99M class—you know who you are. And last, but not least, I would like to thank my parents for their everlasting support.

John M. Kos

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Modern digital communication techniques often require the generation of pseudorandom numbers or sequences. The ability to quickly and easily produce various codes such as maximal length codes, Gold codes, Jet Propulsion Laboratory ranging codes, syncopated codes, and non-linear codes in a laboratory environment is essential. This thesis addresses the issue of providing automated computer control to previously built, manually controlled hardware incorporating the Stanford Telecom STEL-1032 Pseudo-Random Number (PRN) Coder. By incorporating a microcontroller into existing hardware, the STEL-1032 can now be conveniently controlled from a MATLAB® Graphical User Interface (GUI). The user can quickly create, save, and recall various setups for the STEL-1032 in an easy to use GUI environment. In addition to having complete control of the STEL-1032's internal actions, the microcontroller adds an extra measure of control possibilities by using various signals as possible interrupt sources. The microcontroller can sample the STEL-1032's various outputs at a rate up to 320 kHz and the data can be imported directly into MATLAB® for further analysis.

GRAPHICAL USER INTERFACE AND MICROPROCESSOR CONTROL ENHANCEMENT OF A PSEUDORANDOM CODE GENERATOR

CHAPTER 1 INTRODUCTION

1.1 Background

From research efforts during World War II sprang the concept of Spread Spectrum (SS) communications. Early systems used a Transmitted Reference (TR) spreading signal, meaning that the spreading signal was actually transmitted "openly" on a channel separate from the data channel. While this meant that the spreading signal could be truly random, it was not the most secure technique. A TR system also requires twice the bandwidth and transmitting power [1].

A move was made to using a Stored Reference (SR) system in which the spreading code is generated in both the receiver and the transmitter but never actually transmitted. By use of synchronization methods, the receiver code generator is matched to the received signal. Thus, the data can be properly de-spread [1].

Most modern communication, navigation, and radar systems use a SR spread spectrum technique for the generation and transmission of signals. The spreading code is often a pseudorandom code produced by means of one or a combination of several Linear Feedback Shift Registers (LFSR). By using various configurations, an LFSR can generate a plethora of binary sequences such as maximal length, non-linear, Gold, and Jet Propulsion Laboratory (JPL) ranging codes.

Each of these code types provide certain useful characteristics. For example, Gold codes are useful for multiple-access applications. JPL ranging codes provide very long sequences for attaining unmistakable ranging over long distances. Non-linear codes provide the most security.

1.2 Problem Statement

Modern communication and navigation systems use pseudorandom codes for the transfer of information. This research focused on designing and building a system, including software and hardware, capable of quickly, easily, and repeatedly producing various pseudorandom codes. Adding a system capable of giving such ease and versatility of control to the existing hardware built by Brendle [2] was a goal of this thesis.

1.3 Assumptions

This work assumes that the existing hardware would be used as the base upon which to build the new control system. The assumption was also made that all-out speed was not the ultimate goal. Instead, the ability to demonstrate automated control of the Stanford Telecom STEL-1032 Pseudo-Random Number (PRN) Coder was desired.

1.4 Scope

While this system has some incredible code generation capabilities, limitations on the codes tested had to be enforced due to time constraints. The ability to generate various codes was demonstrated, but further evaluation of those codes was not carried out. Chapter 4 presents several test codes and system timing capability test results.

1.5 Approach

After reviewing several options, such as using a digital input/output card or the Personal Computer (PC) parallel port, a design specification was established. A microcontroller was chosen to command the Stanford Telecom STEL-1032. The microcontroller serves as the link between the Stanford Telecom STEL-1032 and the PC Graphical User Interface (GUI). MATLAB® Version 5.1 was chosen as the GUI development platform to provide for easy analysis of collected data.

1.6 Materials and Equipment

The software for the GUI and communications with the microcontroller were developed in MATLAB® Version 5.1, from The Mathworks Inc., Natick, MA. The microcontroller code was first written in C using the READS166 v3.0 development environment from Rigel Corporation, Gainsville, FL. The microcontroller, model RMB-166 v1.1, is also from Rigel Corporation and has 256 kbytes of onboard RAM. Several small parts were supplied by the Air Force Research Laboratory Sensors Directorate. Of course, the Stanford Telecom STEL-1032 PRN Coder was also used.

1.7 Thesis Organization

Chapter 2 presents background information on linear feedback shift registers, pseudorandom noise sequences, the manual control designed and built by Brendle, the concept of a microcontroller, and the concept of the Graphical User Interface. Chapter 3 gives an overview of the Stanford Telecom STEL-1032, proposes some possible metrics for evaluating the performance of the system, explains the coding philosophy used to develop this software, presents a top level view of the software's function, and describes the hardware development. Chapter 4 presents system performance test results such as the maximum data sample rate and interrupt service response time. Conclusions and recommendations on future development are given in Chapter 5. Appendices B, C, and D list the code developed for this thesis. Finally, Appendix E is a short User's Manual to help the user get started.

CHAPTER 2

THEORY

2.1 Introduction

This chapter provides an overview of linear feedback shift registers, some of the various codes used in Spread Spectrum (SS) communications, the manual control previously available, and introduces the microcontroller and Graphical User Interface (GUI) concepts.

2.2 Linear Feedback Shift Registers

Linear Feedback Shift Registers (LFSR) are an effective way of producing a myriad of binary sequences. LFSRs consist of shift registers and modulo-2 addition units. With each clock cycle, the register contents are shifted towards the output end and the new "input" bit value is dictated by the tap connections to the modulo-2 adder and the contents of the associated registers. Figure 2-1 shows an example LFSR.

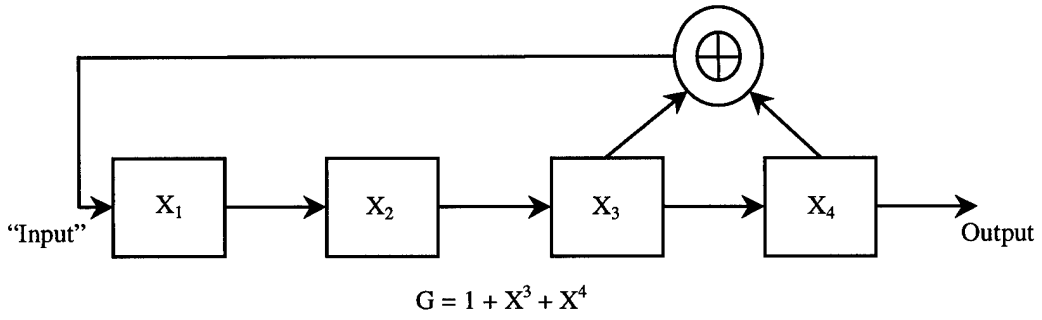


Figure 2-1 Example of Linear Feedback Shift Register [1].

The *generator polynomial*, $G = 1 + X_3 + X_4$, indicates that taps are connected from stages X_3 and X_4 to the modulo-two adder. The *state* of the register is defined as the current contents of the shift registers. For example, if we initially filled X_1 with a one and X_2 through X_4 with zeros, the state would be 1 0 0 0. Based on the generator polynomial and fill indicated

above, the states of this LFSR would be as shown in Table 2-1, below. Notice that after a prescribed number of shifts (15 in this case), the original state reappears and the output sequence starts over.

Table 2-1 States and Output of Example LFSR.

State	X_1	X_2	X_3	X_4	Output
0	1	0	0	0	
1	0	1	0	0	0
2	0	0	1	0	0
3	1	0	0	1	0
4	1	1	0	0	1
5	0	1	1	0	0
6	1	0	1	1	0
7	0	1	0	1	1
8	1	0	1	0	1
9	1	1	0	1	0
10	1	1	1	0	1
11	1	1	1	1	0
12	0	1	1	1	1
13	0	0	1	1	1
14	0	0	0	1	1
15	1	0	0	0	1

2.3 Pseudorandom Codes

The three main categories of pseudorandom codes are *maximal length*, *composite*, and *non-linear*. This section briefly describes each category and gives some typical examples of each. But first, a quick look at the properties of a pseudorandom code.

In order to have statistical properties as near to a truly random signal as possible, a pseudorandom code must meet three basic properties. The first property, *balance*, dictates that the number of ones contained in one period of the sequence should differ from the number of zeros by a maximum of one. The second property, *run*, states that the number of single one or zero runs (a run is defined as a subsequence of like symbols [3]) should be about one-half the total number of runs, double zero or one runs should be about one-quarter the total number, triple runs should be about one-eighth, etc. Finally, the *correlation property* imposes that if the

sequence is cyclically shifted and compared with itself term by term, the number of disagreements and agreements should be within one regardless of the number of shifts [1].

2.3.1 Maximal Length Sequences

A *maximal length sequence* (*m*-sequence) is generated by choosing a suitable generator polynomial, termed a *primitive polynomial*, for the LFSR. The length of the binary sequence, N, is $2^r - 1$, where r is the number of stages, or registers, in the LFSR. An *m-sequence* satisfies all three pseudorandom code properties—balance, run, and correlation--plus two additional properties. The first additional property is the *shift-and-add property* which states that the modulo-2 sum of any *m*-sequence and a phase shifted version of the same sequence produces yet another phase of the same sequence. The second additional property states that the LFSR encounters every possible state except the all zero state before repeating [3]. The example given in Section 2.2, Figure 2-1 and Table 2-1, produces an *m-sequence*.

2.3.2 Composite Codes

Composite codes are produced by combining two or more linear codes, such as those produced by LFSRs. Very complex codes can be generated by this method and can be designed to possess certain useful properties. The next two subsections briefly introduce Gold codes and Jet Propulsion Laboratory ranging codes, two specific forms of composite codes.

2.3.2.1 Gold Codes

The thrust behind Gold code development was to find sets of compatible codes for multiple-access Spread Spectrum (SS) communications. Gold codes are produced by modulo-2 addition of *preferred pairs* of equal length *m*-sequences and result in a code of the same length. These preferred pairs possess certain properties relative to each other. Different codes of a set are produced by using different relative phases for one of the LFSRs. The resulting set of codes exhibit well behaved cross-correlation characteristics [3].

A common example of Gold code use is the Coarse/Acquisition (C/A) code of the Global Positioning System (GPS). Each satellite uses two 10 stage LFSRs to produce its individual code as shown in Figure 2-2. Every satellite has the same two LFSR configurations (generator polynomials) which are all started with an all ones fill. The individual satellite codes are produced by using different phase taps on the G_2 LFSR, effectively using different phases of the code.

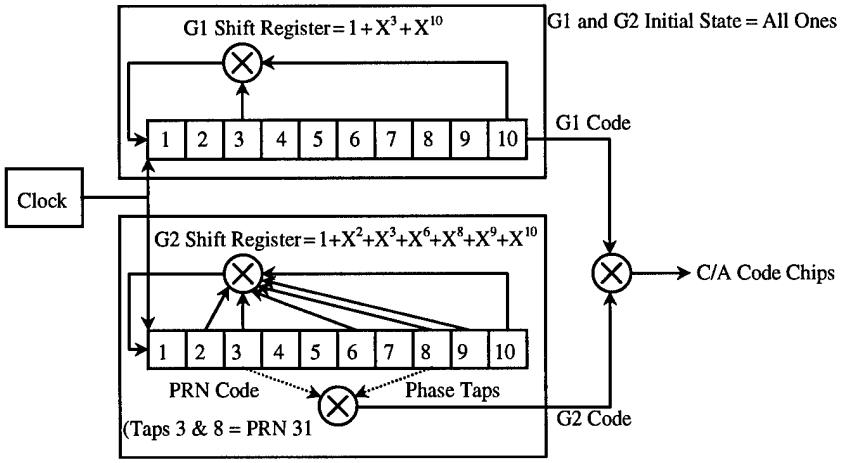


Figure 2-2 Global Positioning System Gold Code Generator Example [4].

2.3.2.2 Jet Propulsion Laboratory Ranging Codes

Jet Propulsion Laboratory (JPL) ranging codes are produced by modulo-2 addition of two or more *m*-sequences of relatively prime lengths. The resulting code has a very long period, allowing for the measurement of long distances. Synchronization with the received signal is also made easier by the fact that each of the component codes can be synchronized independently of the others, thereby, drastically reducing the search space [5].

Although JPL ranging codes and Gold codes can be generated in very similar manners, the difference between them is the length of their component codes. Figure 2-3 gives an example setup for generating a JPL ranging code. Notice that the resulting code length is the product of the component code lengths [5].

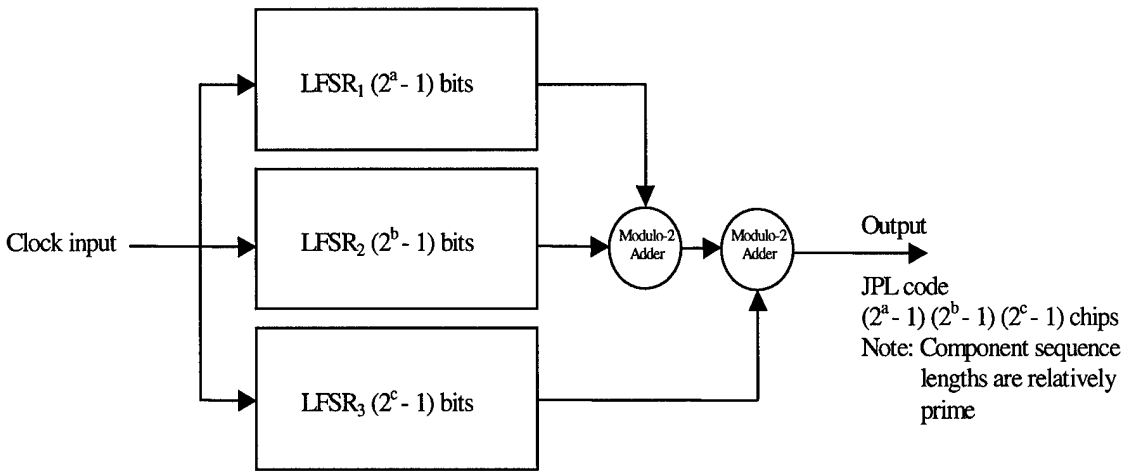


Figure 2-3 JPL Code Generation Example.

2.3.3 Non-Linear Codes

Although maximal-length linear codes are easily produced by Linear Feedback Shift Registers (LFSRs), they are not very secure. For example, algorithms such as Berelekamp-Massey exist for reverse engineering the generator polynomial if $2r$ successive code symbols are known, where r is the length of the original LFSR. Since it is often desirable to keep digital communications private, other code generation methods have been developed which make determining the spreading code more difficult for an unintended receiver because they can not be described by linear recurrence relationships [3]. It should be noted that more than one definition of a "non-linear" code exists. For example, some definitions are broad enough to include any code created by the modulo-2 addition of two or more codes.

One method of generating a non-linear code is using two or more linear codes as the index for a look-up table. The look-up table can be constructed in such a manner that the resulting output cannot be written as a linear combination of the original codes. Such a method is shown in Figure 2-4.

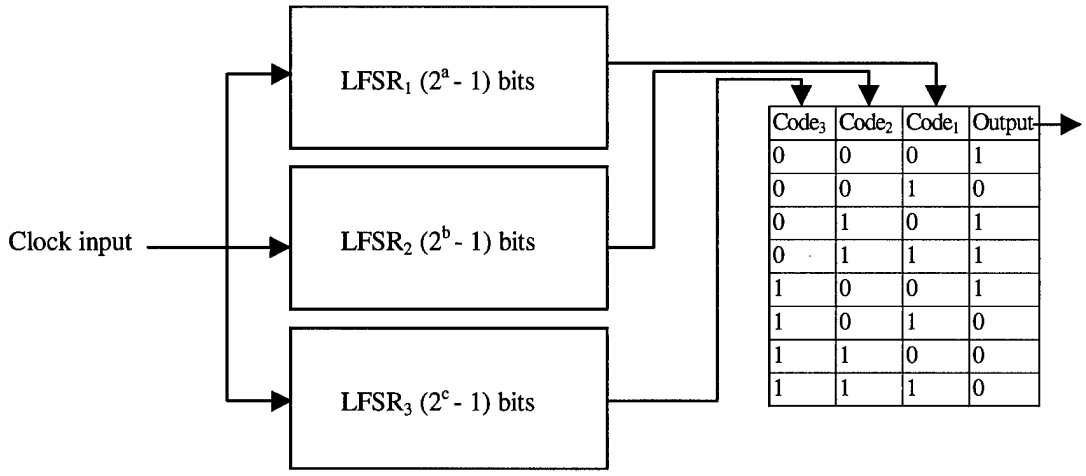


Figure 2-4 Example of Non-Linear Code Generation Using a Lookup Table.

2.4 Manual Control

Previous work by Brendle provides for manual control of the Stanford Telecom STEL-1032 [2]. By using thumbwheels and pushbuttons, the user can set the STEL-1032 registers, load the coders, and enable/disable the clock inputs. The hardware also provides BNC connectors for all inputs and outputs. The outputs are buffered and capable of driving 50 Ω loads. The circuit design included partial provisions for the future addition of automated control. For more complete design information, the reader is directed to Ref. 2. The modifications made to this hardware are discussed in Chapter 3.

2.5 Microcontroller

Microcontrollers are the control system of the present and future. They can be found in everything from toasters to automobiles to jet aircraft. Microcontrollers often reduce complex hardware control issues down to connecting a few sensors and writing some code. By quickly establishing a hardware design, the code can then be written and rewritten until the desired effects are achieved without ever changing the hardware.

A microcontroller is basically a one chip computer typically containing a central processing unit, random access memory and/or read only memory, input and output units, timers,

and an interrupt controller. They often require very little, if any, supporting hardware to form fully functioning control systems. They are available in devices with anywhere from 16 pins with only a few input/output lines to hundreds of pins with many input/output lines, from relatively slow clock rates to relatively fast clock rates, from a memory space counted in bytes to megabytes, etc [6].

2.6 Graphical User Interface

If you have used a Windows, Icon, Menu, and Pointer (WIMP) interface such as Microsoft Windows®95, then you have used a Graphical User Interface (GUI). The GUI concept provides a visual interface which is reasonably intuitive to a human user, thus, acting as a gateway to the underlying software's capabilities.

2.7 Summary

This chapter provides an overview of linear feedback shift registers, some of the various codes used in spread spectrum communications, the manual control previously available, and introduces the microcontroller and Graphical User Interface (GUI) concepts. It was shown that non-linear codes and composite codes such as Gold codes and Jet Propulsion Laboratory ranging codes can be produced by combining multiple linear feedback shift registers in prescribed manners.

CHAPTER 3

SYSTEM DESIGN METHODOLOGY

3.1 Introduction

The purpose of this thesis was to design and build a user friendly, efficient means for producing various pseudorandom sequences. This goal was accomplished by adding automated (computer) control to a previously developed manual interface to the Stanford Telecom STEL-1032 Pseudo-Random Number Coder [2]. By using a premanufactured, commercially available microcontroller evaluation board, hardware design time was converted to software design time.

The microcontroller acts as an interface between the MATLAB® Graphical User Interface running on a Personal Computer (PC) and the STEL-1032. The PC and microcontroller are connected by a standard RS-232 serial interface. The microcontroller interconnects with the STEL-1032 control inputs through manual/auto control selection logic. Several STEL-1032 outputs and inputs are connected to the microcontroller for data collection and additional control possibilities.

This chapter outlines the STEL-1032 architecture, describes the microcontroller selection process, reviews the existing hardware design, and explains the hardware modification and software design methodology.

3.2 STEL-1032 Pseudo-Random Number (PRN) Coder Overview

The Stanford Telecom STEL-1032 Pseudo-Random Number (PRN) Coder contains three independent, 32-stage Linear Feedback Shift Registers (LFSRs) labeled $Coder_0$, $Coder_1$, and $Coder_2$. Each coder is capable of generating codes up to $2^{32} - 1$ (4,294,967,295) chips in length at

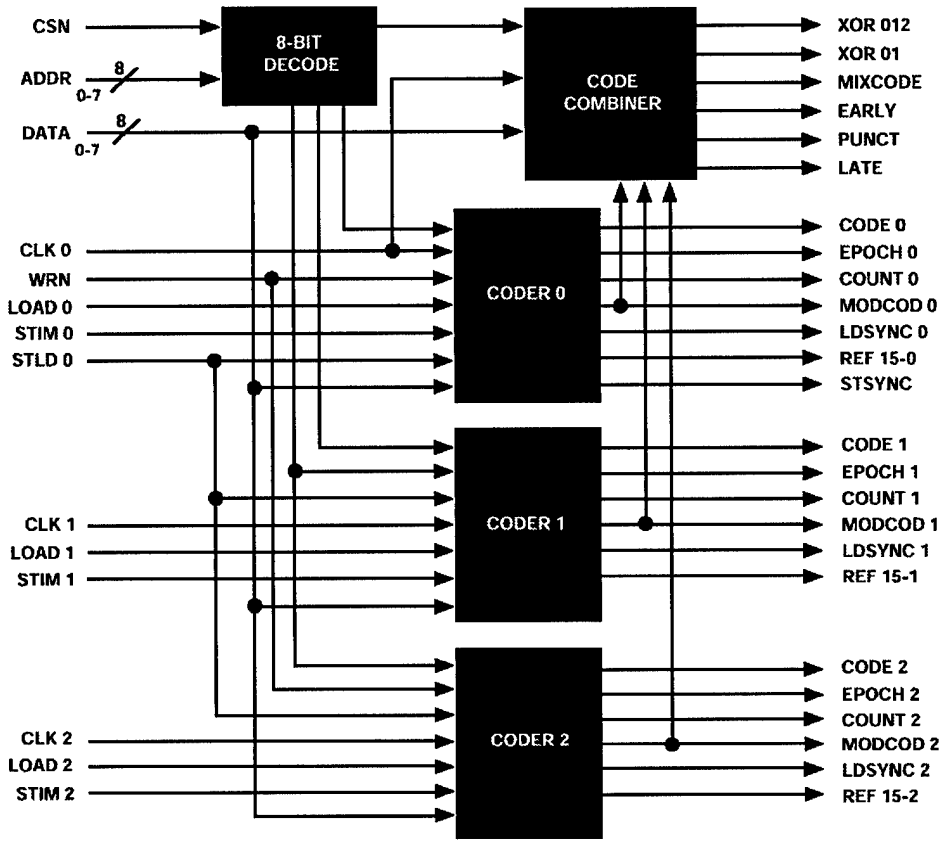


Figure 3-1 STEL-1032 Block Diagram [7].

chip rates up to 30 MHz. The STEL-1032 also contains provisions for code modulation, modulo-2 addition of various internal signals, and non-linear code generation via a lookup table [7].

The next four subsections outline the inputs, outputs, registers, and control mechanisms of the STEL-1032. The overall block diagram is shown in Figure 3-1. For a more detailed description, the reader is directed to Ref. 7, the Stanford Telecom data sheet for the STEL-1032. Unfortunately, the STEL-1032 is out of production.

3.2.1 PRN Coder Inputs

The Stanford Telecom STEL-1032 has a total of 29 input lines. These lines and a brief description of their associated functions are listed below in Table 3-1. Note that the "0-2" subscripts, such as on $LOAD_{0\text{-}2}$, indicate that three LOAD input lines exist such that $LOAD_0$ is associated with $Coder_0$, $LOAD_1$ is associated with $Coder_1$, and so forth.

Table 3-1 STEL-1032 Inputs [7].

Input	Function
RESET	Resets all registers to zero
CSN	Enables loading of data when low
WRN	Data is written into register when low and latched on the rising edge
$ADDR_{0-4}$	Selects target register for a write
$ADDR_{5-7}$	Coder and code combiner selection for a write
$DATA_{0-7}$	Data to be written into the register
$LOAD_{0-2}$	Transfer contents of INIT register to the PRN register and COUNT to the Counter register.
CLK_{0-2}	Clock inputs to Coders 0-2
$STIM_{0-2}$	Data to be modulo-2 added to $CODE_{0-2}$
STLD	Latches data on $STIM_{0-2}$

3.2.2 PRN Coder Outputs

The STEL-1032 has 25 output lines, as described in Table 3-2. Note that the "0-2" subscripts, such as on $CODE_{0-2}$, indicate that three CODE output lines exist such that $CODE_0$ is associated with $Coder_0$, $CODE_1$ is associated with $Coder_1$, and so forth.

Table 3-2 STEL-1032 Outputs [7].

Output	Function
$CODE_{0-2}$	PRN generator outputs from the register bit selected by the Phase Mux register
$MODCOD_{0-2}$	$CODE_{0-2}$ output after modulation by the $STIM_{0-2}$ input
PUNCT	MIXCODE delayed by one clock cycle
EARLY	Half clock cycle delay of MIXCODE
LATE	Half clock cycle delay of PUNCT
$LDSYNC_{0-7}$	Low when INIT register transferred to PRN Generator
$REF15_{0-2}$	Code output derived from fifteenth register bit
STSYNC	Low for one clock cycle after the STLD input signal
$EPOCH_{0-2}$	Low when PRN Generator equals the EPOCH register
$COUNT_{0-2}$	Delayed replica of $LDSYNC_{0-2}$ output
XOR01	Modulo-2 addition of $CODE_0$ and $CODE_1$
XOR012	Modulo-2 addition of $CODE_0$, $CODE_1$, and $CODE_2$
MIXCODE	Bit stored in Code Combiner Lookup Register as addressed by $MODCOD_{0-2}$

3.2.3 Registers

Each coder of the STEL-1032 has four 32-bit registers, one eight bit register, and one five bit register. The four 32-bit registers are the MASK, INIT, EPOCH, and COUNT registers. The eight bit register is the CTL register and the five bit register is the MUX register. There is one eight bit register common to the three coders, the Code Combiner Lookup register [7].

The *MASK* register contains a 32-bit binary representation of the generator polynomial. If the polynomial is expressed as

$$G = 1 + D_1(x) + D_2(x^2) + D_3(x^3)...+D_{31}(x^{31}) + D_{32}(x^{32}),$$

then D_1 = Bit 0, D_2 = Bit 1, D_3 = Bit 2, and so forth. Notice that the STEL-1032's binary representation does not include the D_0 term which is assumed to always be one, as is necessary for an LFSR. Changes to the MASK register affect the PRN generator immediately [7].

The 32-bit *INIT* register is used to establish the initial fill, or phase, of the generated code. The INIT register's contents are transferred to the PRN Generator upon a load command, either from the external LOAD input line or from the internal control logic [7].

The contents of the PRN Generator's registers are constantly compared to the contents of the 32-bit *EPOCH* register. Upon a match, the EPOCH output line goes low for the duration of the match. All bits of the EPOCH register, beyond the MASK register's most significant bit, must be zero for proper operation [7].

The 32-bit *COUNT* register's value is transferred to the counter upon a load command, either from the LOAD input or from the internal control logic [7].

The five bit Phase *MUX* register selects which PRN Generator register tap is connected to the CODE output line. A value of zero selects tap one and a value of 31 selects tap 32 [7].

The eight bit *CTL* register determines how the coders respond to EPOCH and COUNT pulses [7]. Table 3-3 outlines the CTL register options.

Table 3-3 STEL-1032 Control Register Bit Pair Functions [7].

B7 B6	Function
0 0	Counter is not reloaded on any EPOCH pulse
0 1	Counter is reloaded on $EPOCH_0$ pulse
1 0	Counter is reloaded on $EPOCH_1$ pulse
1 1	Counter is reloaded on $EPOCH_2$ pulse
B5 B4	**Function**
0 0	Counter is not reloaded on any COUNT pulse
0 1	Counter is reloaded on $COUNT_0$ pulse
1 0	Counter is reloaded on $COUNT_1$ pulse
1 1	Counter is reloaded on $COUNT_2$ pulse
B3 B2	**Function**
0 0	PRN Gen. is not reloaded on any EPOCH pulse
0 1	PRN Gen. is reloaded on $EPOCH_0$ pulse
1 0	PRN Gen. is reloaded on $EPOCH_1$ pulse
1 1	PRN Gen. is reloaded on $EPOCH_2$ pulse
B1 B0	**Function**
0 0	PRN Gen. is not reloaded on any COUNT pulse
0 1	PRN Gen. is reloaded on $COUNT_0$ pulse
1 0	PRN Gen. is reloaded on $COUNT_1$ pulse
1 1	PRN Gen. is reloaded on $COUNT_2$ pulse

Table 3-4 Register Address Selection and Coder Selection [7].

A4 A3 A2 A1 A0	Register Selected
0 0 0 0 0	MASK Register bits 0-7
0 0 0 0 1	MASK Register bits 8-15
0 0 0 1 0	MASK Register bits 16-23
0 0 0 1 1	MASK Register bits 24-31
0 0 1 0 0	INIT Register bits 0-7
0 0 1 0 1	INIT Register bits 8-15
0 0 1 1 0	INIT Register bits 16-23
0 0 1 1 1	INIT Register bits 24-31
0 1 0 0 0	EPOCH Register bits 0-7
0 1 0 0 1	EPOCH Register bits 8-15
0 1 0 1 0	EPOCH Register bits 16-23
0 1 0 1 1	EPOCH Register bits 24-31
0 1 1 0 0	COUNT Register bits 0-7
0 1 1 0 1	COUNT Register bits 8-15
0 1 1 1 0	COUNT Register bits 16-23
0 1 1 1 1	COUNT Register bits 24-31
1 0 0 0 0	MUX Register bits 0-4
1 0 0 0 1	CTL Register bits 0-7

A7 A6 A5	Coder Selected
0 0 0	$Coder_0$
0 0 1	$Coder_1$
0 1 0	$Coder_2$
0 1 1	All coders
1 0 0	Code Combiner Lookup register

3.2.4Control

Control of the STEL-1032 is achieved by loading the internal registers, as outlined in the Registers section above, with appropriate values. Loading the registers is accomplished by using the eight bit address and eight bit data input lines. Assuming the Chip Select (CSN) is low and the correct address and data have been established on appropriate lines, the Register Write control (WRN) is given a low pulse. Data is latched into the register on the rising edge of the WRN signal. Table 3-4 outlines the addressing scheme.

3.3 Microcontroller Selection

Thousands of microcontroller variations exist from manufacturers such as Intel, Siemens, Atmel, Parallax, Motorola, Philips, Texas Instruments, and Dallas Semiconductor. Likewise, thousands of microcontroller evaluation boards are available from these manufacturers, as well as, other commercial companies. By applying a set of selection criteria, one product was singled out for this effort. The selection criteria included:

1. Adequate input/output capabilities
2. Adequate performance prospects
3. Adequate memory
4. Usability of development environment
5. Large base of on-line (World Wide Web) support
6. Cost
7. Cost to performance ratio
8. Available on a ready-to-go, commercial evaluation board

Two examples of contenders which did not make the final cut are the Tecny Electronics G200-22 Model and the Rigel Corporation RMB-167i. The Tecny G200-22 sports an 8-bit Intel 80C51GB microcontroller running at 22 MHz with 32 kbytes of RAM and costs about $75 [8]. While this board sounds inviting at first, it simply does not have enough memory, input/output pins, or speed. Thus, it does not satisfy several of the "adequate" criteria listed above. On the other hand, the Rigel RMB-167i has a 16-bit Siemens 80C167 microcontroller running at 40 MHz and is available with 256 kbytes of RAM and costs $210 [9]. Although the Siemens

80C167 has an increased number of input/output pins and some additional instructions, as compared to the Siemens 80C166, they both have the same computing power. Even though the RMB-167i comes in above all the "adequate" bars, the cost and cost-to-performance ratio are not favorable. Although the Tecny G200 and Rigel Corp. RMB-167i are fine products, they do not satisfy project requirements.

While searching for possible products, it was quickly concluded that price alone is not an indicator of performance. Usually a higher price simply means that more software is included or that the product is aimed at the educational market.

The final product selected is Rigel Corporation's RMB-166 with 256 kbytes of RAM, costing approximately $130. The microcontroller on this board is the Siemens 80C166 16-bit high-performance microcontroller running at 40 MHz. The package included a demo version of the READS166 v3.00 Integrated Development Environment with Rc66 Compiler/Debugger which proved to be adequate for this study.

3.4 Hardware Design

The previously built hardware included partial provisions for adding automated computer control [2]. Buffers were in place for the eight address lines and the eight data lines. However, no interfaces to the Chip Select (CSN), Register Write (WRN), Reset, or the $LOAD_{0-2}$ lines were included.

To accomplish integration of the Rigel Corp. RMB-166 microcontroller into the existing hardware, three main circuits were developed and added to the hardware. The first serves as the address decoder logic and controls the STEL-1032 WRN input. The second circuit provides an interface to the CSN, WRN, RESET, and $LOAD_{0-2}$ control lines. The third circuit adds buffering for three additional external inputs.

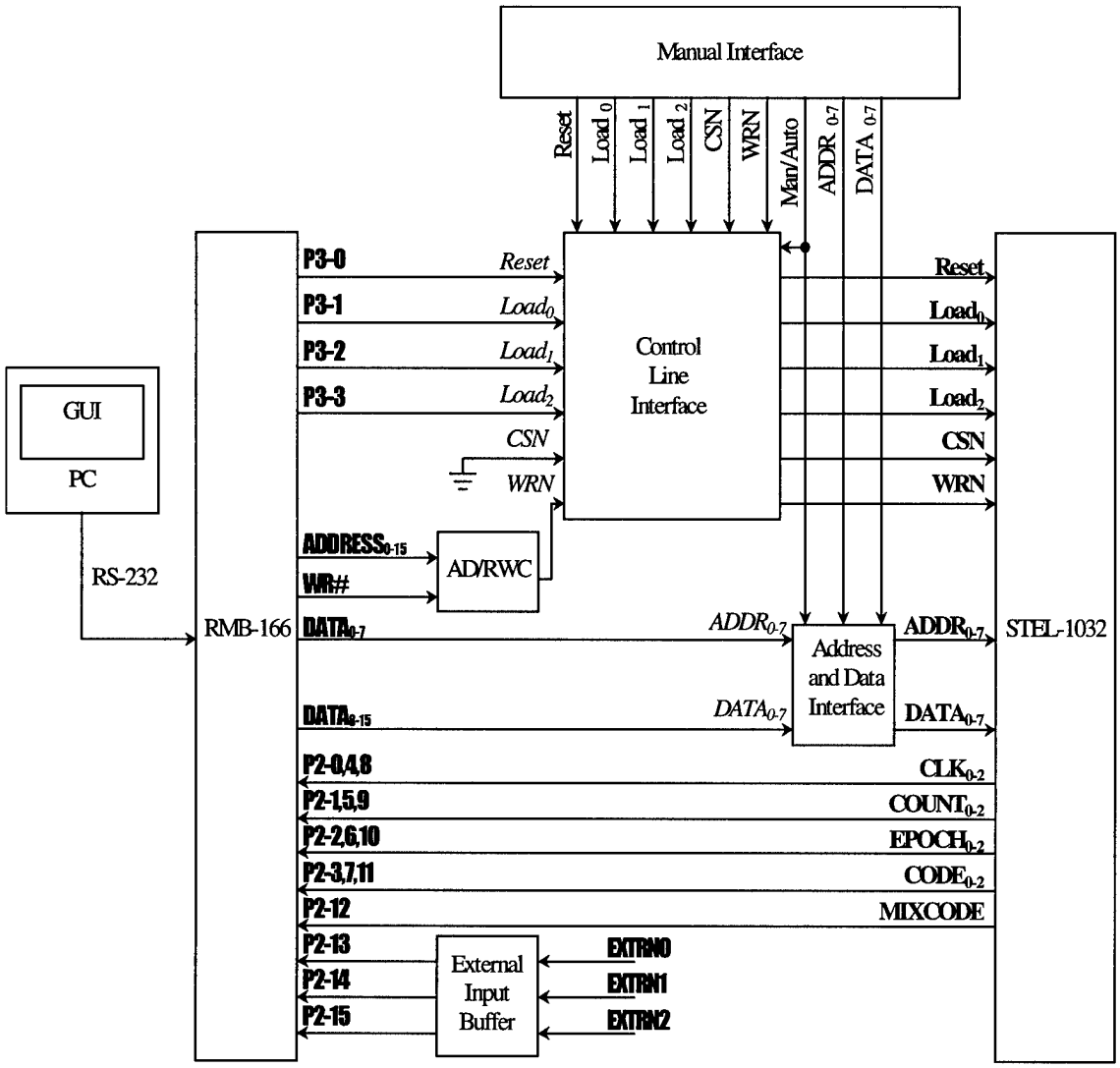


Figure 3-2 Block Diagram of System Layout.

Figure 3-2 gives an overall view of the new control circuit layout. In this figure, the signals are "coded" by font style. **Impact** style font indicates output lines of the RMB-166. *Times New Roman Italics* specify translation of the **Impact** lines to STEL-1032 input lines. Times New Roman normal designates the lines coming from the manual control section. Finally, **Times New Roman Bold** lines are actual STEL-1032 input control lines. The Address and Data Interface block was already available in the hardware.

3.4.1 Address Decoder/Register Write Control

The *Address Decoder/Register Write Control* (AD/RWC) circuit produces a low pulse when two conditions are met, 1) the address present on the microcontroller's $ADDRESS_{0\text{-}15}$ lines match a preset address, and 2) the microcontroller's WR# line is low. The preset address is $F9FE_{16}$ (hexadecimal) which is the last word address available on the external memory bus.

The AD/RWC output is connected to the STEL-1032's WRN input through the Control Line Interface circuit as shown in Figure 3-2. Thus, when data is written to address $F9FE_{16}$, the data present on the microcontroller's $DATA_{8\text{-}15}$ lines is written to the STEL-1032 register addressed by the microcontroller's $DATA_{0\text{-}7}$ lines.

The circuit consists of two 74LS688 8-bit Identity Comparators as shown in Figure 3-3.

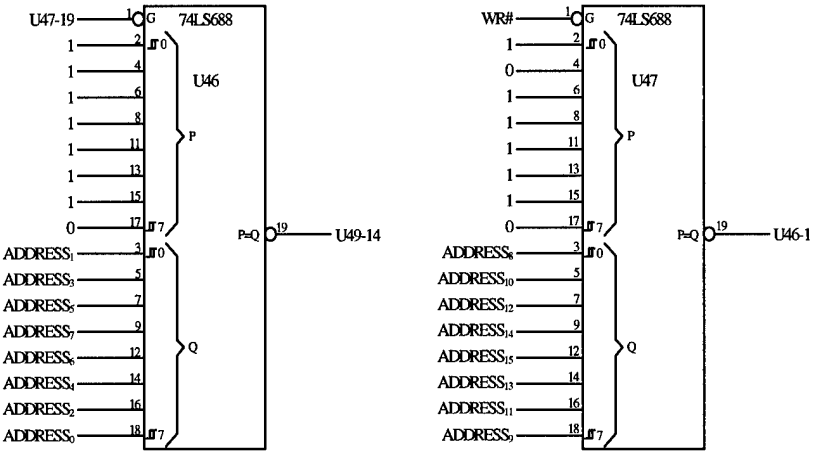


Figure 3-3 Address Decoder/Register Write Control (AD/RWC) Circuit.

3.4.2 Control Line Interface

The *Control Line Interface* permits the RMB-166 to control the RESET, CSN, WRN, $LOAD_0$, $LOAD_1$, and $LOAD_2$ input lines of the STEL-1032. The circuit uses two 74LS157 Quadruple 2-Line to 1-Line Data Selectors and is shown in Figure 3-4.

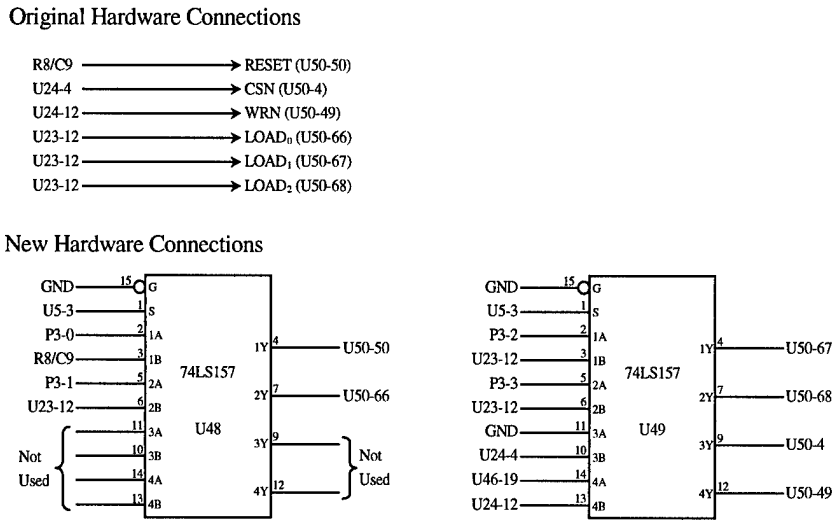


Figure 3-4 Control Line Interface Circuit.

3.4.3 External Input Buffer

The *External Input Buffer* circuit merely provides protection between the external inputs and the RMB-166 inputs while also aiding in noise immunity with the 74LS14's input hysteresis. Figure 3-5 shows the circuit diagram.

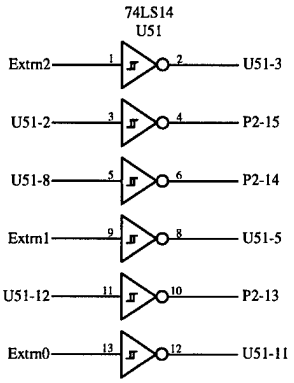


Figure 3-5 External Input Buffer Circuit.

3.4.4 Interrupt Trigger/Data Collection

Ten STEL-1032 outputs, three STEL-1032 inputs, and three external inputs are connected directly to Port 2 of the RMB-166. These inputs (with respect to the RMB-166) are

capable of triggering events and being sampled, as is covered in the Software Design section to come. Table 3-5 outlines the signal connections to the RMB-166.

Table 3-5 Interrupt Trigger/Data Collection Connections.

Signal	STEL-1032 Input/Output	RMB-166 Input	Signal	STEL-1032 Input/Output	RMB-166 Input
CLK_0	I	P2-0	CLK_2	I	P2-8
$COUNT_0$	O	P2-1	$COUNT_2$	O	P2-9
$EPOCH_0$	O	P2-2	$EPOCH_2$	O	P2-10
$CODE_0$	O	P2-3	$CODE_2$	O	P2-11
CLK_1	I	P2-4	MIXCODE	O	P2-12
$COUNT_1$	O	P2-5	$EXTRN_0$		P2-13
$EPOCH_1$	O	P2-6	$EXTRN_1$		P2-14
$CODE_1$	O	P2-7	$EXTRN_2$		P2-15

3.5 Software Design

The software is broken down into three parts: the MATLAB® Graphical User Interfaces (GUIs), the microcontroller "resident" code, and the "user" code. The user develops a STEL-1032 setup using the GUIs, then downloads the resulting user code to the microcontroller via the resident code. The basic event flow is depicted in Figure 3-6. The user may bypass the Process List Quick Builder and start with the Process List Developer.

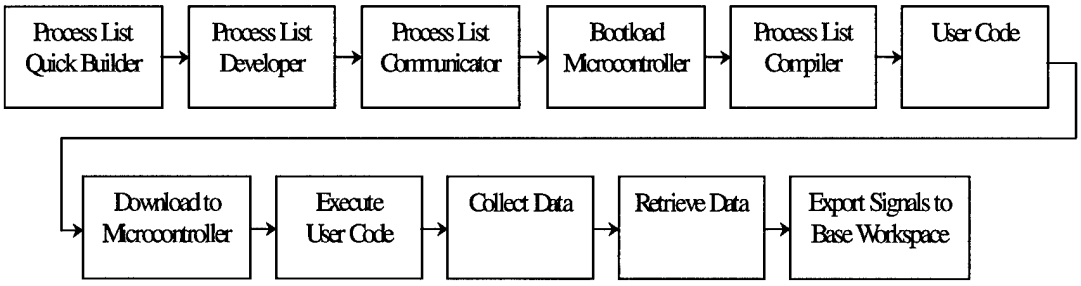


Figure 3-6 Flow of Events.

3.5.1 Microcontroller "Resident" Code

The microcontroller "resident" code consists of helper functions which handle such activities as downloading and executing the "user" code, transmitting collected data, starting the

collection process, stopping the collection process, resetting the system, and sending an Identification (ID) string. This resident code is downloaded to the microcontroller and executed by the MATLAB® GUI after a hardware system reset (pressing the hardware Reset button on the front of the box).

The resident code was originally written in C and compiled using Rigel Corporation's READS166 v3.00 Integrated Development Environment with Rc66 Compiler/Debugger. Although the resulting assembly language code was operationally correct, it was determined to be very inefficient and much larger than necessary. Assembling the Rc66 Compiler's assembly code produced machine language code approximately 4.6 kbytes in size. Manual coding reduced the machine language code to 994 bytes while actually enhancing functionality. The Siemens 80C166 assembly language resident code is listed in Appendix C.

In the following subsections, the notation "[NL]" is introduced and stands for ASCII new line character, number 10. The term "received" refers to receiving data from the PC through the serial port and "return(s)" or "transmit(s)" refers to sending data to the serial port. When "xxxx" or "yyyy" appears, it indicates a hexadecimal number in its ASCII representation. For example, $10BA_{16}$, is transmitted as the sequence [character 49][character 48][character 66][character 65].

3.5.1.1 Identification Routine

When the ASCII character "I" is received, the *identification routine* sends the string "BK v1.0 Loaded[NL]" to the serial port.

3.5.1.2 Download Routine

The *download routine* receives the user code and places it in the proper memory locations. This transfer is initiated when "Dxxxx" is received, where "xxxx" is the number of bytes to download in hexadecimal. The resident code first returns "DOWN[NL]" and then is ready to except the download. When the prescribed number of bytes have been received, the

microcontroller transmits the address after the last byte of user code in hexadecimal followed by "[NL]."

3.5.1.3 Execute Routine

The reception of the "E" character causes the resident code to respond with "EXEC[NL]" and then execute the user code.

3.5.1.4 Start Collection Routine

The *start collection routine* starts data collection on the indicated clock signal and takes the indicated number of samples. Data collection is initiated upon receiving "Cxxxxw" where "xxxx" is the number of samples to take in hexadecimal and "w" is which clock to trigger on ("0" is CLK_0, "1" is CLK_1, and "2" is CLK_2). The microcontroller responds with "CLCT[NL]," or *collect*, indicating that a correct collect command was received.

Upon completing data collection, the microcontroller transmits "FINC[NL]," or *finished collecting*.

3.5.1.5 Send Routine

Upon receiving "Sxxxx," the microcontroller responds with "SENDyyyy" and proceeds to transmit "yyyy" bytes of data. The number "yyyy" is the lesser of either two times "xxxx" or two times the number of data samples which have actually been collected.

3.5.1.6 Stop All Routine

The *stop all routine* disables all interrupts and then responds with "STPA[NL]," or *stop all*. It is called when "X" is received.

3.5.1.7 Finish Collection Routine

Upon reception of "F," data collection is halted. The microcontroller responds with "STPC[NL]," or *stop collection*, and then "xxxx[NL]" where "xxxx" is twice the number of samples taken in hexadecimal.

3.5.1.8 Reset Routine

The *reset routine* disables all interrupts, stops the collection process, resets the collection variables, and resets the user code. Receiving "R" calls the reset routine. The microcontroller's response is "STPA[NL]" followed by "RST[NL]," or *reset*.

3.5.2 MATLAB® Graphical User Interface

The MATLAB® part of this system has three parts: the *Process List Quick Builder*, the *Process List Developer*, and the *Process List Communicator*. Each of these pieces work together to make a user friendly environment for developing setups for the STEL-1032. The MATLAB® code developed for this thesis is included in Appendix B.

3.5.2.1 Process List Quick Builder

The *Process List Quick Builder* (PLQB), as shown in Figure 3-7, allows the user to access all the Stanford Telecom STEL-1032's internal registers with one easy to use GUI. When the user has completed setting the registers, all values are transferred to the Process List Developer (PLD) by first clicking the *Validate* button.

The *Validate* button checks all values to insure that they are between acceptable limits. If a value is determined to be an invalid entry, the associated label turns red. It returns to black when the value is corrected and the *Validate* button is pressed again.

The checkboxes next to the "Coder 1," "Coder 2," and "Mixcode" labels are used to indicate the use of the respective STEL-1032 sections. The values of any section with an unchecked box are not exported to the PLD.

Once validation is complete, the *Export to PLD* button is used to transfer all values from the identified (checked) sections to the PLD in the form of a Process List (PL). If the PLD is not already open, it opens automatically.

The user can save and recall specific setups using the "Workspace" menu item on the menu bar.

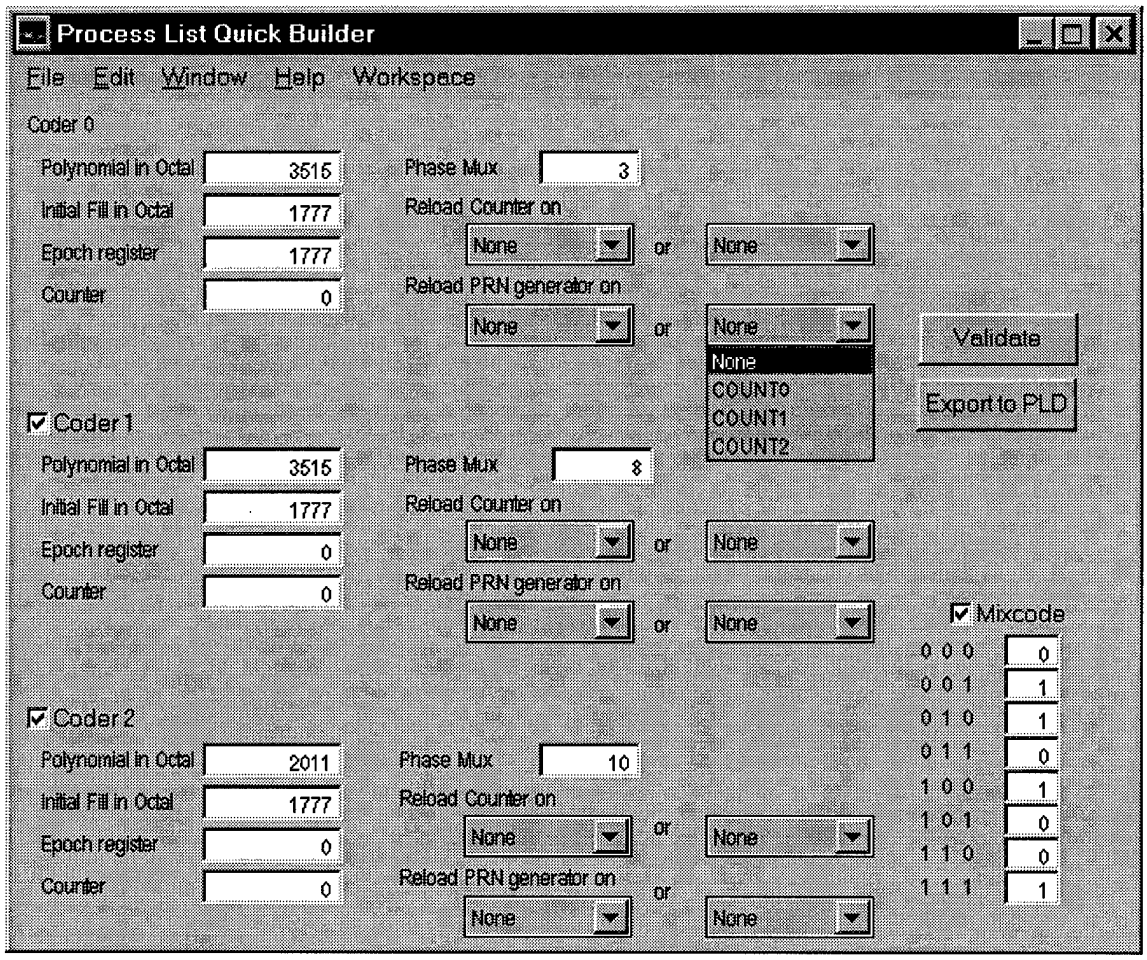


Figure 3-7 Process List Quick Builder Screen Shot.

3.5.2.2 Process List Developer

From the *Process List Developer* (PLD) GUI, shown in Figure 3-8, the user has access to all capabilities of the system, including setting the STEL-1032 registers, setting up Interrupt Service Routines (ISRs), initiating data collection, and terminating data collection. Using the

Process List Item Builder (PLIB) of the PLD, the user can add *Process List Items* (PLIs) to the *Process List* (PL). The PL represents the pseudo-code which is compiled and downloaded to the microcontroller.

Under the "List" menu item, the user can save and recall PLs and reset the current PL. The "Comment" menu selection allows the user to view and edit the *Process List Comment* associated with the current PL. The "Undo" menu contains a list of all deleted PLIs from which the user can reinsert deleted PLIs by simply selecting it on the menu.

By selecting a PLI and pressing the *Edit* button, the user can edit the current PLI using the PLIB. The change is made to the PLI in the PL when the *Add* button is pressed. If any object on the PLD screen is clicked (other than in the PLIB) between pressing the *Edit* button and the *Add* button, edit mode is aborted. As implied, the *Delete* button deletes the currently selected PLI and places it in the Undo menu. Because event order can be important, the user can move a PLI up or down the list using the *Move Up* and *Move Down* buttons. When the user has finished PL development, the *Done* button is pressed which automatically opens the Process List Communicator if it is not already open.

The PLIB consists of three list boxes, a *Value* edit box, and the *Add* button. The first list box, *When*, selects the trigger. When the indicated trigger event occurs, the PLIs associated with it in the PL are executed. All initialization actions are associated with the INIT trigger, which is the first thing executed. The second list box, *Action*, indicates what should be done to the item designated in the third list box, *What*. The *Value* edit box determines what value is associated with the *What* item, if necessary. If no value is associated with the selected *What* item, the *Value* edit box does not appear. The *Base* indicator automatically changes to the base (octal, decimal, or binary) appropriate for the value of the selected *What* item.

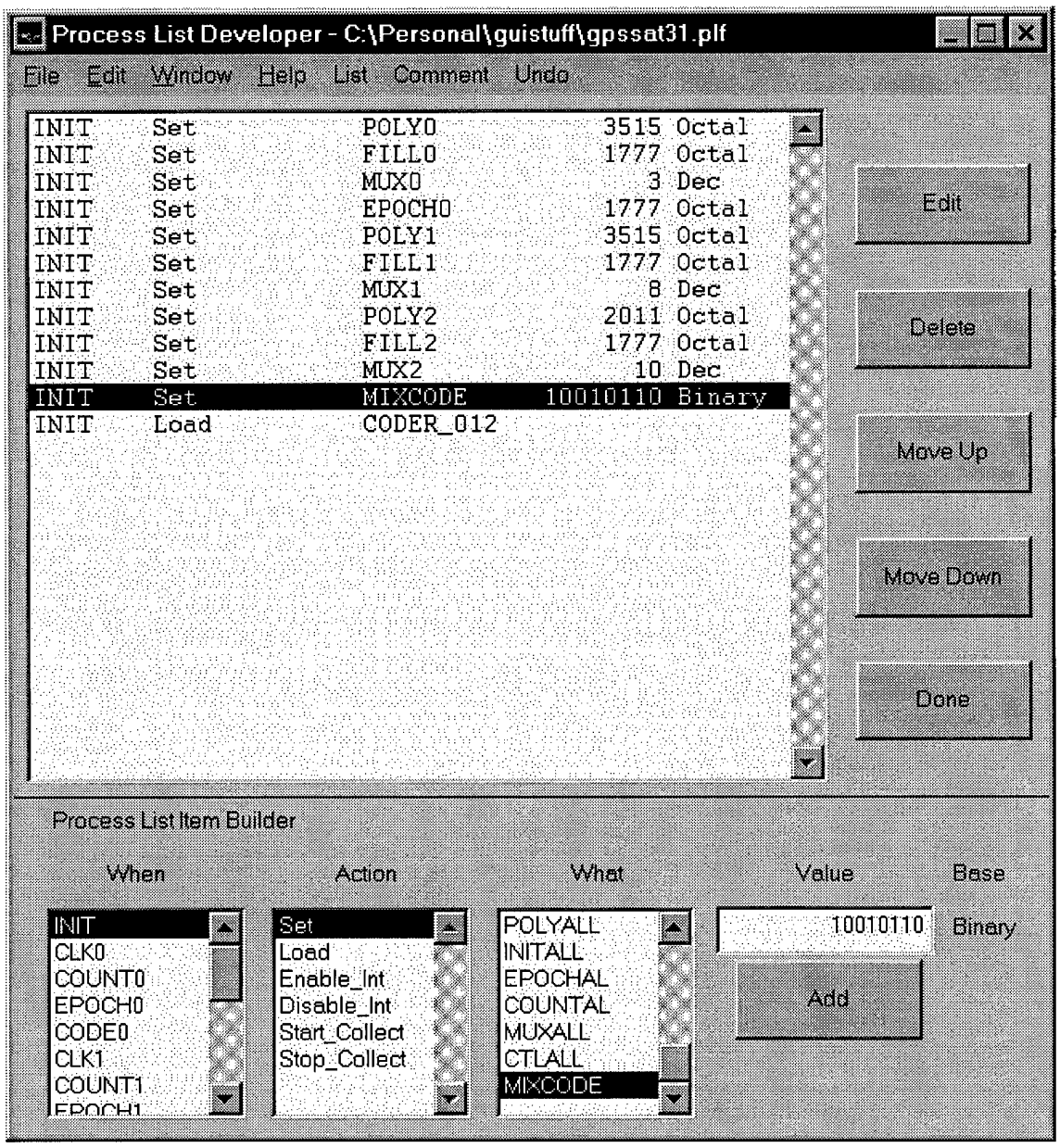


Figure 3-8 Process List Developer Screen Shot.

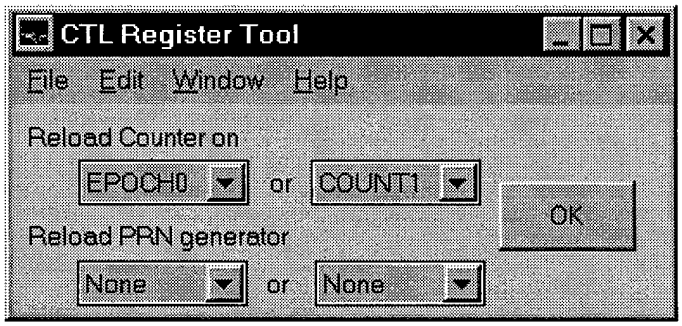


Figure 3-9 Control Register Tool Screen Shot.

When CTL0, CTL1, CTL2, or CTLALL (the STEL-1032 control registers) is selected in the *What* list box, clicking on the *Value* edit box brings up the *Control Register Tool* as shown in Figure 3-9. This tool allows the user to set all valid options for the selected CTL register. Clicking the *OK* button transfers the correct binary representation of the user's selections to the *Value* edit box.

3.5.2.3 Process List Communicator

The *Process List Communicator* (PLC), as depicted in Figure 3-10, is the control center for interacting with the microcontroller. The PLC is used to bootload the microcontroller, compile and transfer the PL to the microcontroller, execute the PL, start and stop data collection, stop all actions on the microcontroller, retrieve collected data, and export the various signals to the base MATLAB® workspace.

When the PLC is first started, the *Establish Comm* button is the only option available, all others are grayed out and non-functional. To establish communications and bootload the microcontroller, the user momentarily presses the hardware Reset button on the box and then presses the *Establish Comm* button on the PLC. Upon a successful bootload and execution, the *Establish Comm* button is grayed out and the text "Communication Established" appears next to it. The *Export PL* button also becomes available.

To transfer the PL shown in the PLD GUI to the microcontroller, the user presses the *Export PL* button. This button initiates the compilation of the PL into Siemens 80C166 machine language using the special compiler written for this thesis. The compiled code is then downloaded to the microcontroller. Upon a successful download, the *Execute PL*, *Start Collection*, *Stop Collection*, *Stop All*, and *Retrieve Data* buttons become available, and the text "PL Export Successful" appears next to the *Export PL* button. Execution of the PL occurs when the *Execute PL* button is clicked.

The user can initiate data collection by entering the number of samples to acquire and indicating which clock to use as the sample trigger. When the *Start Collection* button is pressed, data collection begins. Likewise, the user can halt data collection, whether initiated by the *Start Collection* button or by the execution of the PL, by clicking the *Stop Collection* button.

After data collection is complete or halted by the user, the *Retrieve Data* pushbutton is used to upload collected data from the microcontroller. After selecting (checking) and naming (manually typing) the desired signals, pressing the *Export Data* button causes the designated signals to be assigned in the MATLAB® workspace as their user provided names.

As labeled, the *Stop All* button disables *all* interrupt service routines within the microcontroller, including data collection.

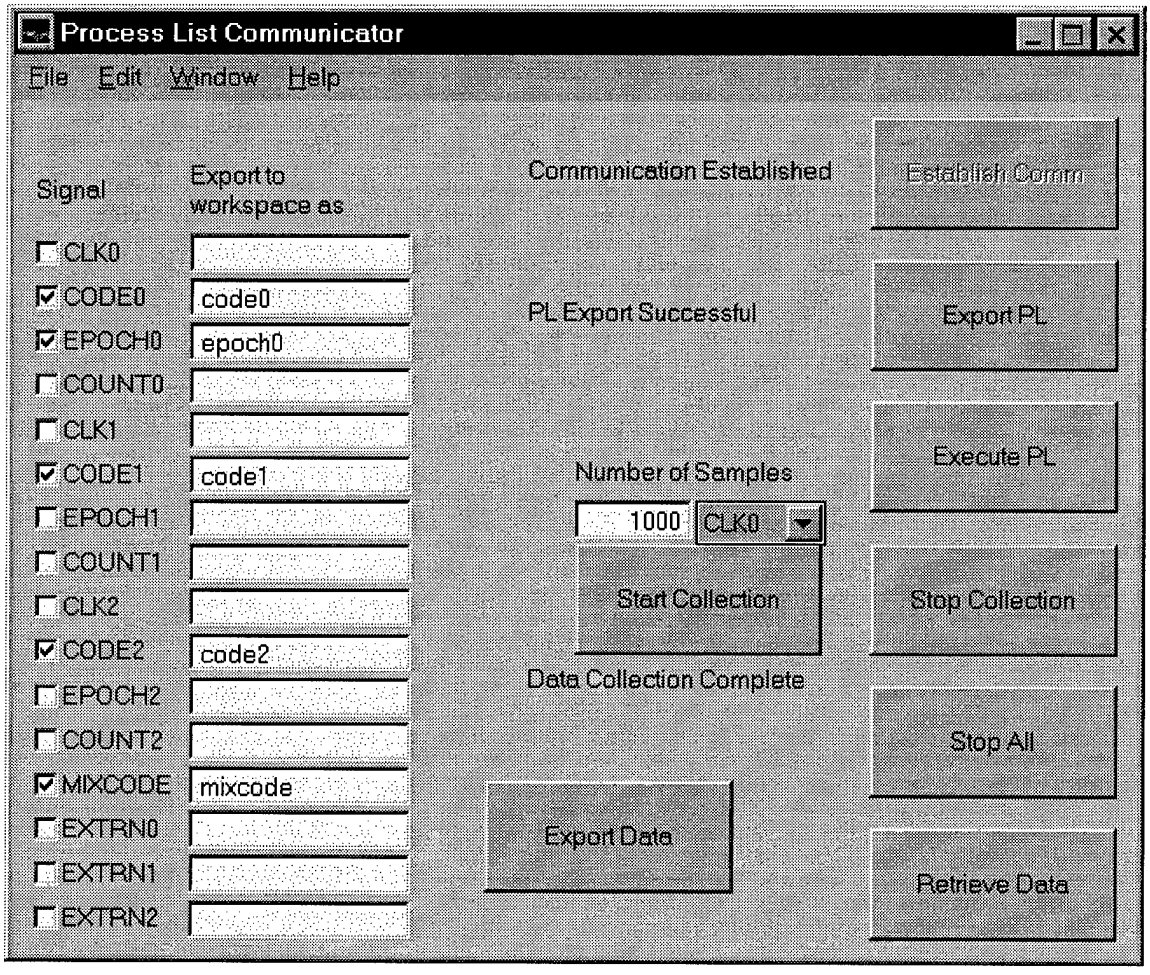


Figure 3-10 Process List Communicator (PLC) Screen Shot.

3.5.3 Microcontroller "User" Code

The "user" code is the result of compiling the PL. This code is Siemens 80C166 machine language and is ready for download to the microcontroller. However, the user code is not stand-alone; it depends on the resident code for certain operations.

Two MATLAB® m-functions, plcompile.m and plcompile2.m, perform the compilation. These two files use code templates and place the proper values and addresses in their respective positions to create Code Fragments (CFs). The CFs are grouped into Code Segments (CSs), with each CS representing the action to be taken when the respective trigger occurs. For example, the INIT CS contains the code to be executed when the user presses the *Execute PL* button on the PLC GUI. This code structure is indicated in Figure 3-11. If no STEL-1032 register contents are modified within a CS, the data portion for that CS does not exist. See Appendix D for a more complete outline of this code.

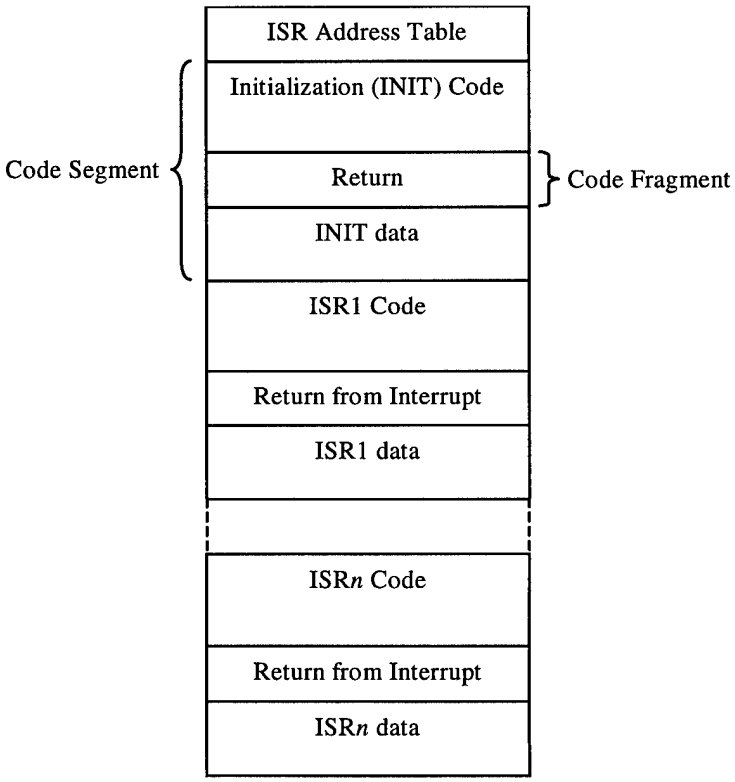


Figure 3-11 "User" Code Structure.

3.6 System Evaluation Metrics

The metrics introduced in this section are used in Chapter 4 to test the system. The main criteria for this system are 1) valid, reliable code production and, 2) efficiency. The metrics include the execution time of all the user code components such that the user can calculate how long a given interrupt service routine should take. The maximum sampling rate is also determined. Finally, the generation of several codes is demonstrated.

3.7 Summary

This chapter provides an overview of the Stanford Telecom STEL-1032's functionality, inputs, and outputs; the microcontroller selection; the hardware design; the software design; and the system evaluation metrics. The STEL-1032 contains three 32-stage, independent pseudo-random number generators. The Rigel Corp. RMB-166 microcontroller was selected based on criteria such as performance capabilities and cost-to-performance ratio. The circuits added to the previous hardware allow the RMB-166 to control the STEL-1032. Software control is provided by three MATLAB® Graphical User Interfaces, the microcontroller "resident" code, and the microcontroller "user" code. Finally, most of the system evaluation metrics are time based.

CHAPTER 4

SYSTEM TEST RESULTS AND ANALYSIS

4.1 Introduction

The previous chapters presented the problem statement, the relevant background information, and the system design methodology. In this chapter, the system performance is evaluated. The two main facets of performance are demonstrated: 1) the system can easily and reliably generate sequences, and 2) all Code Fragment (CF) execution times are measured. By knowing the CF execution times, the user can calculate the total execution time of an Interrupt Service Routine (ISR) and, thus, the maximum rate at which the ISR can be performed.

4.2 System Functionality Validation

Before any sequence can be produced, the Stanford Telecom STEL-1032's registers must be appropriately set and the corresponding coders must be loaded. Figure 4-1 is a screenshot from a LeCroy 9324 Digital Storage Oscilloscope which illustrates a 32-bit register being set in four 8-bit segments by the WRN signal and then the Coder being loaded with the LOAD signal. The Process List (PL) associated with generating this plot is given in Table 4-1.

Table 4-1 Functionality Test Process List.

INIT	Set	POLY0	13 Octal
INIT	Set	FILL0	7 Octal
INIT	Set	EPOCH0	2 Octal
INIT	Enable_Int	EPOCH0	
INIT	Load	CODER0	
EPOCH0	Set	FILL1	34 Octal
EPOCH0	Load	CODER1	

Upon an $EPOCH_0$, the $INIT_1$ register is set to 34_8 and $Coder_1$ is immediately loaded by the $LOAD_1$ signal, as designated by the user.

This one figure indicates the proper operation of many project parts: 1) the user is able to build a PL using the Process List Developer (PLD), 2) the Process List Communicator (PLC) properly bootloads and communicates with the hardware, 3) the "resident" microcontroller code is able to download and execute a PL, 4) the PL Compiler generates correctly functioning code, 5) the Control Line Interface circuit works, and 6) the Address Decoder/Register Write Control circuit operates. These traces could not occur without the proper operation of all project parts.

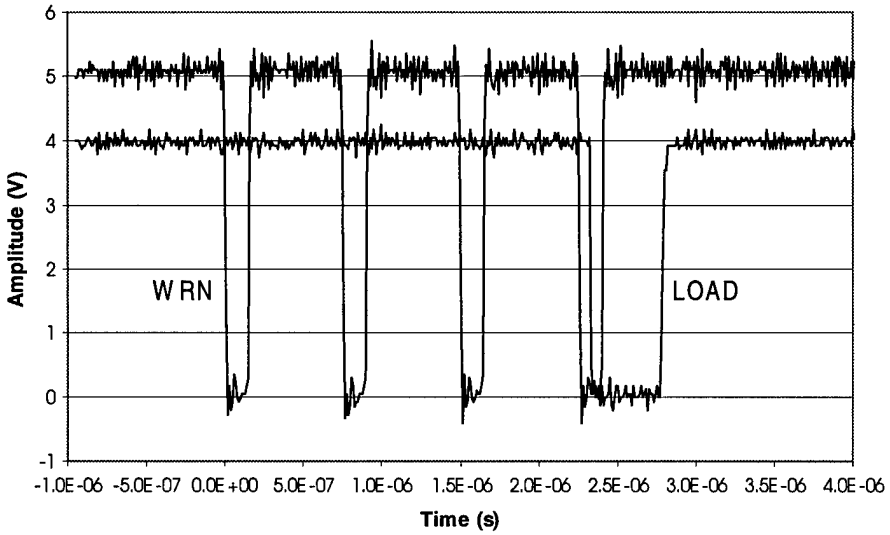


Figure 4-1 32-Bit Register Set Followed by LOAD Pulse.

4.3 Code Production Validation

Proper code production is verified by the successful production of a Global Positioning System (GPS) Coarse/Acquisition (C/A) code, a truncated code, and a short Jet Propulsion Laboratory (JPL) ranging code. Several plots, as well as, the Process Lists used to generate them, appear in the next three subsections.

4.3.1 Global Positioning System (GPS) Coarse/Acquisition (C/A) Code

Production of the GPS satellite number 31 C/A code is possible using the configuration shown in Figure 3-7 and Figure 3-8 in Sections 3.5.2.1 and 3.5.2.2, respectively. The PL is repeated here in Table 4-2. All three coders are clocked synchronously.

Table 4-2 PL to Produce Coarse/Acquisition Code for Satellite Number 31.

```
INIT    Set         POLY0          3515 Octal
INIT    Set         FILL0          1777 Octal
INIT    Set         MUX0              3 Dec
INIT    Set         EPOCH0         1777 Octal
INIT    Set         POLY1          3515 Octal
INIT    Set         FILL1          1777 Octal
INIT    Set         MUX1              8 Dec
INIT    Set         POLY2          2011 Octal
INIT    Set         FILL2          1777 Octal
INIT    Set         MUX2             10 Dec
INIT    Set         MIXCODE    10010110 Binary
INIT    Load        CODER_012
```

Although not the most efficient means of producing C/A code, this intentionally inefficient setup demonstrates the use of all three Coders and the Mixcode. The 1023-chip sequence was generated at a rate of 200 kcps and collected by the microcontroller. In all cases, the collected signal matched the expected output. The beginning of the sequence is shown in Figure 4-2. The top trace, the C/A code, is the exclusive-or (modulo-2 sum) of the other three coder output traces. It should be noted that the exclusive-or output is delayed by one clock cycle.

4.3.2 Truncated Code

Reloading the Coder before the full sequence length is generated results in a truncated code. The $CODE_0$ output shown in Figure 4-3 is such a code produced by the PL of Table 4-3. The generator polynomial 23_8 would normally produce a maximal length sequence (*m*-sequence) 15 bits in length. However, in this setup the code restarts after only 12 clock cycles due to the $LOAD_0$ pulse of the $EPOCH_0$ Interrupt Service Routine (ISR) as shown by the PL in Table 4-3.

The Stanford Telecom STEL-1032 has the capability to perform this vary action on its own, but for demonstration purposes an ISR is used.

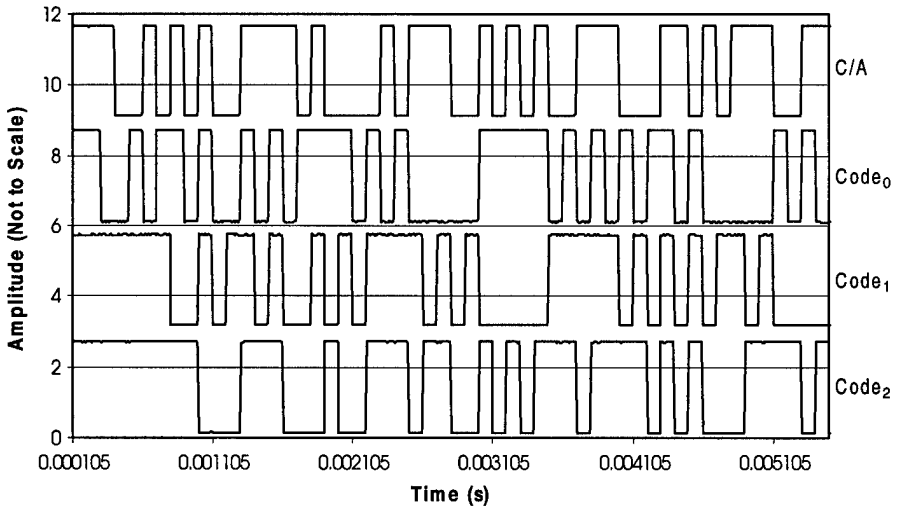


Figure 4-2 GPS C/A Code Number 31.

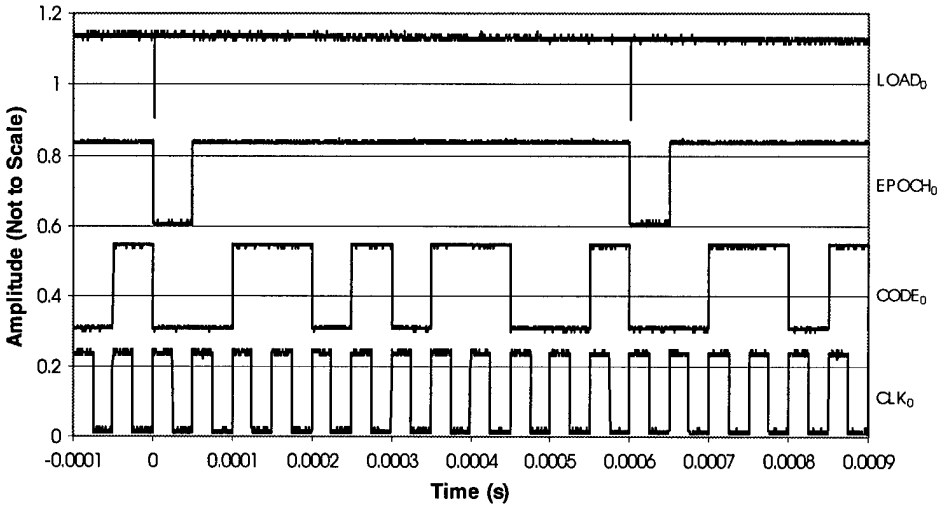


Figure 4-3 Truncated Code Production.

Table 4-3 Process List for Truncated Code Test.

INIT	Set	POLY0	23 Octal
INIT	Set	FILL0	7 Octal
INIT	Set	EPOCH0	2 Octal
INIT	Enable_Int	EPOCH0	
INIT	Load	CODER0	
EPOCH0	Load	CODER0	

Figure 4-4 depicts the inherent delay between the $EPOCH_0$ signal and the ISR response. In this case, the delay is approximately 1.93 μs. The sloped appearance of the $EPOCH_0$, $CODE_0$, and CLK_0 traces is due to averaging several traces on the oscilloscope. Averaging produces a sloped transition because the Coder clock signal, in this case CLK_0, and the microcontroller clock are uncorrelated and, thus, timing jitter occurs.

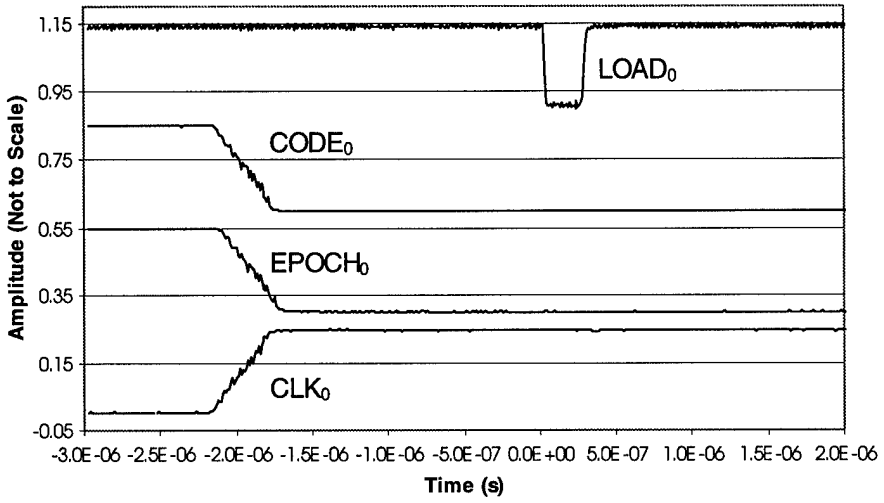


Figure 4-4 Close up of Truncated Code Signals.

4.3.3 Jet Propulsion Laboratory (JPL) Ranging Code

For demonstration purposes, the PL shown in Table 4-4 generates a code having a length equal to the product of the component code lengths. In this case, the component codes have lengths of 2^2 - 1 (3) and 2^3 – 1 (7) which are relatively prime, as is required for the multiplicative length behavior to occur. Thus, the composite code resulting from the modulo-2 addition of the two codes has a length of 21 chips (or bits). Two periods of the composite code are shown in Figure 4-5 where the top trace (XOR_{01}) is the composite code, the middle two are the component codes, and the bottom trace is the clock. The composite code is delayed by one clock cycle due to operational characteristics of the STEL-1032. While such a short code generated in this manner has little practical purpose, correct operation of the hardware is clearly demonstrated.

Table 4-4 Addition of Relatively Prime Length Sequences.

INIT	Set	POLY0	13 Octal
INIT	Set	FILL0	7 Octal
INIT	Set	EPOCH0	7 Octal
INIT	Set	POLY1	7 Octal
INIT	Set	FILL1	3 Octal
INIT	Set	EPOCH1	3 Octal
INIT	Load	CODER_0_1	

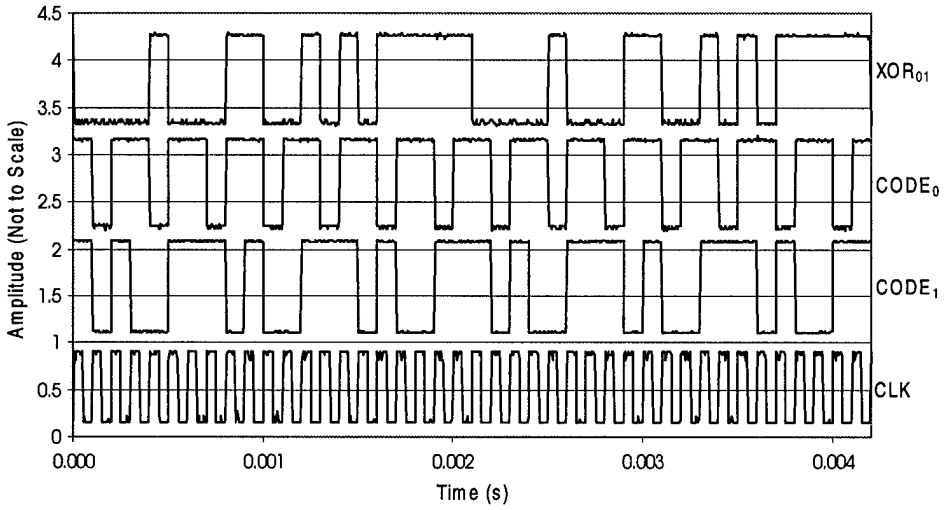


Figure 4-5 Modulo-2 Addition of Codes with Relatively Prime Lengths.

4.4 System Timing Tests

The timing tests are performed by specifying an ISR triggered on the CLK_0 signal and then determining the maximum clock rate at which the ISR can remain synchronized with the clock. By adding various Code Fragments (CFs) and reestablishing the maximum synchronizable clock rate, the execution time of the additional CF can be calculated. The baseline ISR contains the code for one "Load" line in the PL.

To determine the maximum clock rate, an oscilloscope display, such as that shown in Figure 4-6 with the Load signal as the sweep trigger, is observed while the signal generator clock rate is slowly increased. When the ISR cannot stay synchronized with the clock rate, the clock

trace begins to drift. The frequency just below that at which "lock" is broken indicates the maximum ISR frequency and is recorded as such.

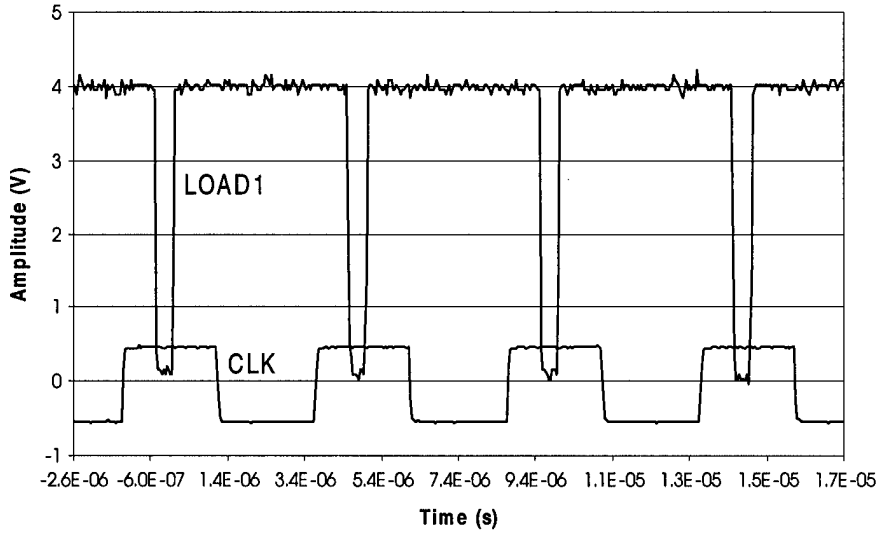


Figure 4-6 LOAD Pulses Shown with Clock Signal.

Table 4-5 Timing Data for "Non-Register" Code Fragments.

ISR Process List Code		Maximum Clock Rate (kHz)
CLK0 Load	CODER1	400.0
CLK0 Load	CODER1	333.3
CLK0 Load	CODER1	
CLK0 Load	CODER1	222.2
CLK0 Load	CODER1	
CLK0 Load	CODER1	
CLK0 Load	CODER1	
CLK0 Load	CODER1	
CLK0 Enable_INT	EPOCH1	333.3
CLK0 Load	CODER1	
CLK0 Enable_INT	EPOCH1	222.2
CLK0 Enable_INT	EPOCH1	
CLK0 Enable_INT	EPOCH1	
CLK0 Enable_INT	EPOCH1	
CLK0 Load	CODER1	
CLK0 Disable_INT	EPOCH1	333.3
CLK0 Load	CODER1	
CLK0 Start_Collect	CLK0	129.0
CLK0 Load	CODER1	
CLK0 Start_Collect	CLK1	117.6
CLK0 Load	CODER1	
CLK0 Start_Collect	CLK2	108.0
CLK0 Load	CODER1	

The data in Table 4-5 were acquired in this fashion for the CFs which do not change any STEL-1032 registers (thus the "non-register" nomenclature). Likewise, data listed in Table 4-6 were collected for CFs which change a STEL-1032 register (thus the "register" designation).

Table 4-6 Timing Data for "Register" Code Fragments.

ISR Process List Code		Maximum Clock Rate (kHz)
CLK0 Set CLK0 Load	POLY1 CODER1	153.8
CLK0 Set CLK0 Set CLK0 Load	POLY1 POLY1 CODER1	105.2
CLK0 Set CLK0 Set CLK0 Set CLK0 Set CLK0 Set CLK0 Load	POLY1 POLY1 POLY1 POLY1 POLY1 CODER1	54.0
CLK0 Set CLK0 Load	MUX1 CODER1	235.2
CLK0 Set CLK0 Set CLK0 Set CLK0 Set CLK0 Set CLK0 Load	MUX 1 MUX 1 MUX1 MUX1 MUX1 CODER1	137.9

4.5 Code Fragment Execution Time Calculation

Using the frequencies listed in the two tables above, the execution time for each type of CF, T_{CF}, is calculated based on the change in maximum frequency when additional CFs are added to the base CF, or

$$T_{CF} = \frac{1}{n}\left(\frac{1}{f_{nCF}} - \frac{1}{f_{BC}}\right) \qquad (4\text{-}1)$$

where n is the number of occurrences of the test CF, f_{nCF} is the maximum frequency with n test CFs included, and f_{BC} is the maximum frequency of the base CF. The purpose of testing with both a single test CF and multiple test CFs is to verify time linearity, as should be the case.

Using Eq. *(4-1)*, the various T_{CF} values are found as listed in Table 4-7. The calculated times agree with theoretical values when the CF machine language and the Siemens 80C166 internal operations are considered.

Table 4-7 Code Fragment Execution Times.

Code Fragment	**Execution Time** (μs)
Set a 32-bit register	$T_{Set32} = 3.0$
Set a 5 or 8-bit register	$T_{Set8} = 0.75$
Load a single Coder	$T_{SLoad} = 0.5$
Load multiple Coders	$T_{MLoad} = 1.0$
Enable an Interrupt Service Routine	$T_{EnableInt} = 0.5$
Disable an Interrupt Service Routine	$T_{DisableInt} = 0.5$
Start CLK_0 Data Collection	$T_{StartCollect0} = 5.25$
Start CLK_1 Data Collection	$T_{StartCollect1} = 6.0$
Start CLK_2 Data Collection	$T_{StartCollect2} = 6.75$
Stop Data Collection	$T_{StopCollect} = 0.5$
"Non-register" Interrupt overhead	$T_{NRIntOverhead} = 2.0$
"Register" Interrupt overhead	$T_{RIntOverhead} = 3.0$

The $T_{NRIntOverhead}$ and $T_{RintOverhead}$ times are determined by the following two equations.

$$T_{RIntOverhead} = \frac{1}{f_{RBC}} - T_{SLoad} - T_{Set8} \qquad (4\text{-}2)$$

$$T_{NRIntOverhead} = \frac{1}{f_{NRBC}} - T_{SLoad} \qquad (4\text{-}3)$$

where f_{RBC} is the maximum rate for a CF consisting of a single Coder Load and an 8-bit register Set and f_{NRBC} is the maximum rate for a CF consisting of a single Coder Load.

4.6 Code Segment Execution Time Calculation

Using the times listed in Table 4-7, the execution time of a Code Segment (CS), or the CFs associated with an Interrupt Service Routine (ISR), can be calculated. Thus, the maximum rate at which a particular ISR can successfully function is determined. The equation for calculating the CS execution time, T_{CS}, in microseconds is

$$T_{CS} = 2 + R + 3N_{Set32} + 0.75N_{Set8} + 0.5\left(N_{SLoad} + N_{EnableInt} + N_{DisableInt} + N_{StopDataCollect}\right) + N_{MLoad} + 5.25N_{StartCollect0} + 6N_{StartCollect1} + 6.75N_{StartCollect2} \quad (4\text{-}4)$$

where $R = 1$ if $(N_{Set32} + N_{Set8}) > 0$, 0 otherwise,
N_{Set8} is the number of 5 or 8-bit registers set,
N_{Set32} is the number of 32-bit registers set,
N_{SLoad} is the number of single Coder loads performed (generally 0 or 1),
N_{MLoad} is the number of multiple Coder loads performed (generally 0 or 1),
$N_{EnableInt}$ is the number of Interrupts enabled,
$N_{DisableInt}$ is the number of Interrupts disabled,
$N_{StopDataCollect}$ is the number of Stop Collects issued,
$N_{StartCollect} = 1$ if $Clock_0$ data collection is initiated, 0 otherwise,
$N_{StartCollect1} = 1$ if $Clock_1$ data collection is initiated, 0 otherwise, and
$N_{StartCollect2} = 1$ if $Clock_2$ data collection is initiated, 0 otherwise.

It follows that the maximum ISR rate is

$$f_{CS} = \frac{1}{T_{CS}} \quad (4\text{-}5)$$

when T_{CS} is converted to seconds.

4.7 Maximum Data Collection Rate

To determine the maximum possible data collection rate with no other ISRs active, $Coder_0$ was configured to produce a fourth order maximal length sequence. While incrementally increasing the code clock rate, or chip rate, 31,000 data samples were obtained at each clock rate. The collected data was compared to the expected result. A point of failure, i.e., the collected code sequence is invalid, was reached at a 340 kHz clock rate. Therefore, the maximum recommended code clock rate for data collection purposes is 320 kHz.

This figure is only valid when no other ISR routines are active. Any other ISR activity will dramatically affect the maximum data collection rate. It should be noted that the designated clock (CLK) signal used for data collection triggering cannot be used for any other ISR purpose.

4.8 Summary

The test results shown in this chapter demonstrate the proper operation of the integrated software and hardware systems by exhibiting the generation of several test codes. The generation

of the test codes confirms that Process Lists are correctly compiled and downloaded to the microcontroller and that the microcontroller properly controls the Stanford Telecom STEL-1032. Tests revealed a maximum data sampling (collection) rate of 320 kHz when no other Interrupt Service Routines (ISRs) are active. All required Code Fragment execution times were determined so that the user may calculate the execution time of any given Code Segment.

CHAPTER 5

CONCLUSIONS AND RECOMMENDATIONS

5.1 Conclusions

This thesis addressed adding automated computer control via a Graphical User Interface (GUI) to a previously built, manually controlled hardware implemented interface to the Telecom STEL-1032 Pseudo-Random Number (PRN) Coder. The MATLAB® GUI allows a user to easily exploit the full capabilities of the STEL-1032 via an efficient, repeatable manner. The user is able to save and recall any number of setups, limited only by available disk space.

Rigel Corporation's RMB-166 microcontroller evaluation board serves as the link between the GUI's running on the Personal Computer (PC) and the STEL-1032. Communications occur between the PC and RMB-166 by way of a standard RS-232 serial interface. The microcontroller is initially bootloaded with "resident" code which handles several basic tasks such as receiving and executing the "user" code. "User" code is generated by compiling the designated Process List, created by the user with the Process List Developer GUI and, optionally, the Process List Quick Builder GUI. All communications with the microcontroller are handled via the Process List Communicator GUI.

5.2 Recommendations for Future Research

Due to the limited scope of this research, several areas related to this work were not addressed. Four areas providing opportunities for further research include:

1. Combining a microcontroller and STEL-1032 into one Application Specific Integrated Circuit (ASIC) or Field Programmable Gate Array (FPGA).
2. Further optimization of the Process List Compiler

3. Further investigation into using the Siemens 80C166 Peripheral Event Controller (PEC) as the data collection agent
4. Modifying the microcontroller code to use the RMB-166's full 256 kbytes of memory

A replacement is necessary for the Stanford Telecom STEL-1032 as it has been out of production for several years. By implementing a microcontroller and a STEL-1032 on a single ASIC or FPGA, many performance issues can be greatly improved. For example, by having 32-bit data paths (or larger), the Coder phase could be changed extremely quickly. The hardware could be optimized to allow fast access to the various registers, providing both greater and quicker control options. The length of the Linear Shift Feedback Registers could also be expanded.

Further optimization of the Process List Compiler could include such issues as only changing the minimum number of 8-bit register chunks as deemed necessary. For instance, if only the 16 least significant bits of the $Coder_0$ MASK register are ever used by the Process List, the 16 most significant bits do not need to be set by any 'Set Poly0" Process List Items.

Using the 80C166's PEC would effectively increase the maximum data collection rate. Expanding the microcontroller's code to use the 256 kbytes of memory would allow for more data samples to be taken.

APPENDIX A

COMMONLY USED ACRONYMS, TERMS, AND SYMBOLS

CF.....Code Fragment: part of CS related to one Process List Item

CS.....Code Segment: contains CFs related to one ISR

GUI.....Graphical User Interface

ISR .....Interrupt Service Routine

LFSR .....Linear Feedback Shift Register

m-sequence.....Maximal length binary PR sequence

PC.....Personal Computer

PL.....Process List

PLC .....Process List Communicator

PLD .....Process List Developer

PLI.....Process List Item

PLIB .....Process List Item Builder

PLQD .....Process List Quick Developer

PRN.....Pseudorandom Number

r.....Number of stages in a LFSR

SS .....Spread Spectrum

APPENDIX B

MATLAB® CODE

This Appendix contains the MATLAB® code for the Graphical User Interfaces (GUIs) developed for this thesis.

B.1 Process List Quick Builder GUI Code

The following MATLAB® script, "plqb.m," creates the GUI for the Process List Quick Builder. Please note that the file "plqb.mat" must also be available.

```
function plqb()
% function plqb()
%  Process List Quick Builder.  Allows the user access to
%  all of the STEL-1032's internal registers on one
%  screen.

load plqb

textFontSize = 0.5;

plqbtext.Units = 'normalized';
plqbtext.HorizontalAlignment = 'left';
plqbtext.BackgroundColor = [0.8 0.8 0.8];
plqbtext.FontUnits = 'points';
plqbtext.FontSize = 8;
plqbtext.FontName = 'Arial Narrow';
plqbtext.Style = 'text';
texth = 0.038;
textw1 = 0.145;

% Properties for the edit boxes
plqbedit.Units = 'normalized';
plqbedit.HorizontalAlignment = 'right';
plqbedit.BackgroundColor = [1 1 1];
plqbedit.FontUnits = 'points';
plqbedit.FontSize = 8;
plqbedit.FontName = 'Arial Narrow';
plqbedit.Style = 'edit';
edith = 0.04;

% Properties for the pop up menus
plqbpopup.Units = 'normalized';
plqbpopup.BackgroundColor = [0.75 0.75 0.75];
plqbpopup.FontUnits = 'points';
plqbpopup.FontSize = 8;
plqbpopup.FontName = 'Arial Narrow';
plqbpopup.Style = 'popupmenu';
```

```
a = figure('Units','pixels', 'Color',[0.8 0.8 0.8], ...
   'Colormap',mat0, 'Name','Process List Quick Builder', ...
   'NumberTitle','off', 'PointerShapeCData',mat1, ...
   'Position',[10 40 640 480], 'HandleVisibility','callback',...
   'Tag','qb');

b = uicontrol('Parent',a, plqbtext, ...
   'Position',[0.013 0.95 textw1 texth], ...
   'String','Coder 0', 'Tag','StaticText6');
b = uicontrol('Parent',a, plqbtext, ...
   'Position',[0.025 0.9 textw1 texth], ...
   'String','Polynomial in Octal', 'Tag','poly0text');
b = uicontrol('Parent',a, plqbtext, ...
   'Position',[0.025 0.85 textw1 texth], ...
   'String','Initial Fill in Octal', 'Tag','fill0text');
b = uicontrol('Parent',a, plqbtext, ...
   'Position',[0.025 0.80 textw1 texth], ...
   'String','Epoch register', 'Tag','epoch0text');
b = uicontrol('Parent',a, plqbtext, ...
   'Position',[0.025 0.75 textw1 texth], ...
   'String','Counter', 'Tag','counter0text');
b = uicontrol('Parent',a, plqbtext, ...
   'Position',[0.349933 0.9 0.10 texth], ...
   'String','Phase Mux', 'Tag','phase_mux0text');
b = uicontrol('Parent',a, plqbtext, ...
   'Position',[0.349933 0.855 0.2 texth], ...
   'String','Reload Counter on', 'Tag','StaticText7');
b = uicontrol('Parent',a, plqbtext, ...
   'Position',[0.572005 0.807 0.0363392 texth], ...
   'String','or', 'Tag','StaticText8');
b = uicontrol('Parent',a, plqbtext, ...
   'Position',[0.349933 0.76 0.2 texth], ...
   'String','Reload PRN generator on', 'Tag','StaticText9');
b = uicontrol('Parent',a, plqbtext, ...
   'Position',[0.572005 0.712 0.0363392 texth], ...
   'String','or', 'Tag','StaticText8');
b = uicontrol('Parent',a, plqbtext, ...
   'Position',[0.025 0.55 textw1 texth], ...
   'String','Polynomial in Octal', 'Tag','poly1text');
b = uicontrol('Parent',a, plqbtext, ...
   'Position',[0.025 0.5 textw1 texth], ...
   'String','Initial Fill in Octal', 'Tag','fill1text');
b = uicontrol('Parent',a, plqbtext, ...
   'Position',[0.025 0.45 textw1 texth], ...
   'String','Epoch register', 'Tag','epoch1text');
b = uicontrol('Parent',a, plqbtext, ...
   'Position',[0.025 0.4 textw1 texth], ...
   'String','Counter', 'Tag','counter1text');
b = uicontrol('Parent',a, plqbtext, ...
   'Position',[0.349933 0.55 0.10 texth], ...
   'String','Phase Mux', 'Tag','phase_mux1text');
b = uicontrol('Parent',a, plqbtext, ...
   'Position',[0.349933 0.505 0.2 texth], ...
   'String','Reload Counter on', 'Tag','StaticText7');
b = uicontrol('Parent',a, plqbtext, ...
   'Position',[0.572005 0.457 0.04 texth], ...
```

```
   'String','or', 'Tag','StaticText8');
b = uicontrol('Parent',a, plqbtext, ...
   'Position',[0.349933 0.41 0.2 texth], ...
   'String','Reload PRN generator on', 'Tag','StaticText9');
b = uicontrol('Parent',a, plqbtext, ...
   'Position',[0.572005 0.362 0.04 texth], ...
   'String','or', 'Tag','StaticText8');
b = uicontrol('Parent',a, plqbtext, ...
   'Position',[0.025 0.20 textw1 texth], ...
   'String','Polynomial in Octal', 'Tag','poly2text');
b = uicontrol('Parent',a, plqbtext, ...
   'Position',[0.025 0.15 textw1 texth], ...
   'String','Initial Fill in Octal', 'Tag','fill2text');
b = uicontrol('Parent',a, plqbtext, ...
   'Position',[0.025 0.1 textw1 texth], ...
   'String','Epoch register', 'Tag','epoch2text');
b = uicontrol('Parent',a, plqbtext, ...
   'Position',[0.025 0.05 textw1 texth], ...
   'String','Counter', 'Tag','counter2text');
b = uicontrol('Parent',a, plqbtext, ...
   'Position',[0.344549 0.20 0.10 texth], ...
   'String','Phase Mux', 'Tag','phase_mux2text');
b = uicontrol('Parent',a, plqbtext, ...
   'Position',[0.344549 0.155 0.2 texth], ...
   'String','Reload Counter on', 'Tag','StaticText7');
b = uicontrol('Parent',a, plqbtext, ...
   'Position',[0.572005 0.121779 0.0363392 texth], ...
   'String','or', 'Tag','StaticText8');
b = uicontrol('Parent',a, plqbtext, ...
   'Position',[0.344549 0.06 0.2 texth], ...
   'String','Reload PRN generator on', 'Tag','StaticText9');
b = uicontrol('Parent',a, plqbtext, ...
   'Position',[0.572005 0.0435583 0.0363392 texth], ...
   'String','or', 'Tag','StaticText8');
b = uicontrol('Parent',a, plqbtext, ...
   'Position',[0.810596 0.33 0.0728477 texth], ...
   'String','0  0  0', 'Tag','StaticText5');
b = uicontrol('Parent',a, plqbtext, ...
   'Position',[0.810596 0.29 0.0728477 texth], ...
   'String','0  0  1', 'Tag','StaticText5');
b = uicontrol('Parent',a, plqbtext, ...
   'Position',[0.810596 0.25 0.0728477 texth], ...
   'String','0  1  0', 'Tag','StaticText5');
b = uicontrol('Parent',a, plqbtext, ...
   'Position',[0.810596 0.21 0.0728477 texth], ...
   'String','0  1  1', 'Tag','StaticText5');
b = uicontrol('Parent',a, plqbtext, ...
   'Position',[0.810596 0.17 0.0728477 texth], ...
   'String','1  0  0', 'Tag','StaticText5');
b = uicontrol('Parent',a, plqbtext, ...
   'Position',[0.810596 0.13 0.0728477 texth], ...
   'String','1  0  1', 'Tag','StaticText5');
b = uicontrol('Parent',a, plqbtext, ...
   'Position',[0.810596 0.09 0.0728477 texth], ...
   'String','1  1  0', 'Tag','StaticText5');
b = uicontrol('Parent',a, plqbtext, ...
   'Position',[0.810596 0.05 0.0728477 texth], ...
```

```
   'String','1  1  1', 'Tag','StaticText5');

b = uicontrol('Parent',a, plqbedit, ...
   'Position',[0.17 0.9 0.125 edith], 'String','23', 'Tag','poly0');
b = uicontrol('Parent',a, plqbedit, ...
   'Position',[0.17 0.85 0.125 edith], 'String','5', 'Tag','fill0');
b = uicontrol('Parent',a, plqbedit, ...
   'Position',[0.17 0.80 0.125 edith], 'String','2', 'Tag','epoch0');
b = uicontrol('Parent',a, plqbedit, ...
   'Position',[0.17 0.75 0.125 edith], ...
   'String','1000', 'Tag','counter0');
b = uicontrol('Parent',a, plqbedit, ...
   'Position',[0.469717 0.9 0.09 edith], ...
   'String','1', 'Tag','phase_mux0');
b = uicontrol('Parent',a, plqbpopup, ...
   'Position',[0.403769 0.825 0.15 0.03], ...
   'String',mat2, 'Tag','counter0epoch', 'Value',1);
b = uicontrol('Parent',a, plqbpopup, ...
   'Position',[0.617766 0.825 0.15 0.03], ...
   'String',mat3, 'Tag','counter0count', 'Value',1);
b = uicontrol('Parent',a, plqbpopup, ...
   'Position',[0.403769 0.73 0.15 0.03], ...
   'String',mat2, 'Tag','fill0epoch', 'Value',1);
b = uicontrol('Parent',a, plqbpopup, ...
   'Position',[0.617766 0.73 0.15 0.03], ...
   'String',mat3, 'Tag','fill0count', 'Value',1);

b = uicontrol('Parent',a, 'Units','normalized', ...
   'FontUnits','normalized', 'BackgroundColor',[0.8 0.8 0.8], ...
   'FontSize',textFontSize, 'HorizontalAlignment','left', ...
   'Position',[0.013 0.60 textw1 texth], 'String','Coder 1', ...
   'Style','checkbox', 'Tag','coder1', ...
   'Value', 0);

b = uicontrol('Parent',a, plqbedit, ...
   'Position',[0.17 0.55 0.125 edith], 'String','23', 'Tag','poly1');
b = uicontrol('Parent',a, plqbedit, ...
   'Position',[0.17 0.5 0.125 edith], 'String','1', 'Tag','fill1');
b = uicontrol('Parent',a, plqbedit, ...
   'Position',[0.17 0.45 0.125 edith], 'String','7', 'Tag','epoch1');
b = uicontrol('Parent',a, plqbedit, ...
   'Position',[0.17 0.4 0.125 edith], 'String','15', 'Tag','counter1');
b = uicontrol('Parent',a, plqbedit, ...
   'Position',[0.480485 0.55 0.09 edith], ...
   'String','1', 'Tag','phase_mux1');
b = uicontrol('Parent',a, plqbpopup, ...
   'Position',[0.403769 0.475 0.15 0.03], ...
   'String',mat2, 'Tag','counter1epoch', 'Value',1);
b = uicontrol('Parent',a, plqbpopup, ...
   'Position',[0.617766 0.475 0.15 0.03], ...
   'String',mat3, 'Tag','counter1count', 'Value',1);
b = uicontrol('Parent',a, plqbpopup, ...
   'Position',[0.403769 0.38 0.15 0.03], ...
   'String',mat2, 'Tag','fill1epoch', 'Value',1);
b = uicontrol('Parent',a, plqbpopup, ...
   'Position',[0.617766 0.38 0.15 0.03], ...
   'String',mat3, 'Tag','fill1count', 'Value',1);
```

```
b = uicontrol('Parent',a, 'Units','normalized', ...
   'FontUnits','normalized', 'BackgroundColor',[0.8 0.8 0.8], ...
   'FontSize',textFontSize, 'HorizontalAlignment','left', ...
   'Position',[0.013 0.25 textw1 texth], 'String','Coder 2', ...
   'Style','checkbox', 'Tag','coder2', ...
   'Value', 0);

b = uicontrol('Parent',a, plqbedit, ...
   'Position',[0.17 0.2 0.125 edith], 'String','23', 'Tag','poly2');
b = uicontrol('Parent',a, plqbedit, ...
   'Position',[0.17 0.15 0.125 edith], 'String','7', 'Tag','fill2');
b = uicontrol('Parent',a, plqbedit, ...
   'Position',[0.17 0.1 0.125 edith], 'String','7', 'Tag','epoch2');
b = uicontrol('Parent',a, plqbedit, ...
   'Position',[0.17 0.05 0.125 edith],'String','15', 'Tag','counter2');
b = uicontrol('Parent',a, plqbedit, ...
   'Position',[0.47 0.2 0.09 edith], 'String','2', 'Tag','phase_mux2');
b = uicontrol('Parent',a, plqbpopup, ...
   'Position',[0.402649 0.125 0.15 0.03], ...
   'String',mat2, 'Tag','counter2epoch', 'Value',1);
b = uicontrol('Parent',a, plqbpopup, ...
   'Position',[0.617219 0.125 0.15 0.03], ...
   'String',mat3, 'Tag','counter2count', 'Value',1);
b = uicontrol('Parent',a, plqbpopup, ...
   'Position',[0.402649 0.03 0.15 0.03], ...
   'String',mat2, 'Tag','fill2epoch', 'Value',1);
b = uicontrol('Parent',a, plqbpopup, ...
   'Position',[0.617219 0.03 0.15 0.03], ...
   'String',mat3, 'Tag','fill2count', 'Value',1);

b = uicontrol('Parent',a, 'Units','normalized', ...
   'FontUnits','normalized', 'BackgroundColor',[0.8 0.8 0.8], ...
   'FontSize',textFontSize, 'HorizontalAlignment','left', ...
   'Position',[0.835 0.375 textw1 texth], 'String','Mixcode', ...
   'Style','checkbox', 'Tag','mixcode', ...
   'Value', 0);

b = uicontrol('Parent',a, plqbedit, ...
    'Position',[0.883444 0.33 0.05 edith],'String','0',...
    'Tag','combiner0');
b = uicontrol('Parent',a, plqbedit, ...
   'Position',[0.883444 0.29 0.05 edith],
   'String','1', 'Tag','combiner1');
b = uicontrol('Parent',a, plqbedit, ...
   'HorizontalAlignment','right', ...
   'Position',[0.883444 0.25 0.05 edith], ...
   'String','0', 'Tag','combiner2');
b = uicontrol('Parent',a, plqbedit, ...
   'Position',[0.883444 0.21 0.05 edith], ...
   'String','1', 'Tag','combiner3');
b = uicontrol('Parent',a, plqbedit, ...
   'Position',[0.883444 0.17 0.05 edith], ...
   'String','0', 'Tag','combiner4');
b = uicontrol('Parent',a, plqbedit, ...
   'Position',[0.883444 0.13 0.05 edith], ...
   'String','1', 'Tag','combiner5');
```

```
b = uicontrol('Parent',a, plqbedit, ...
   'Position',[0.883444 0.09 0.05 edith], ...
   'String','0', 'Tag','combiner6');
b = uicontrol('Parent',a, plqbedit, ...
   'Position',[0.883444 0.05 0.05 edith], ...
   'String','1', 'Tag','combiner7');

b = uicontrol('Parent',a, 'Units','normalized', ...
   'Callback','plqbhelper validate;', ...
   'Position',[0.806623 0.686441 0.143046 0.0627119], ...
   'String','Validate', 'Tag','Pushbutton1');
b = uicontrol('Parent',a, 'Units','normalized', ...
   'Callback','plqb2pld', ...
   'Position',[0.805298 0.610169 0.143046 0.0627119], ...
   'String','Export to PLD', 'Tag','Pushbutton1');

f = uimenu('Parent',a,'Label','Workspace');
uimenu(f,'Label','Open','Callback','plqbhelper open');
uimenu(f,'Label','Save','Callback','plqbhelper save');
```

B.2 Process List Developer GUI Code

The following MATLAB® script, "pld.m" creates the screen for the Process List Developer developed for this research. The files "pld.mat" and "closepld.m" must also be present.

```
function pld(plfile)
% function pld(plfile)
%  Process List Developer allows for the creation and editing of
%  Process Lists for use with the B/K Box.
%  If a file name is not passed in, a blank Process List is created.


load pld
global WHENSTR ACTIONSTR WHATSETSTR WHATINTSTR WHATLOADSTR
global WHATCOLLECTSTR
global LISTH WHENH ACTIONH WHATH VALUEH BASEH
global PLNAME PLCOMMENT PLEDIT PLUNDOH PLSAVED

WHENSTR =     ['INIT    ';'CLK0    ';'COUNT0 ';'EPOCH0 ';'CODE0   ';...
      'CLK1    ';'COUNT1 ';'EPOCH1 ';'CODE1   ';'CLK2    ';'COUNT2 ';...
      'EPOCH2 ';'CODE2   ';'MIXCODE';'EXTRN0 ';'EXTRN1 ';'EXTRN2 '];
ACTIONSTR =  ['Set           ';'Load          ';'Enable_Int    ';...
      'Disable_Int  ';'Start_Collect';'Stop_Collect '];
WHATSETSTR = ['POLY0   ';'FILL0   ';'EPOCH0 ';'COUNT0 '; ...
      'MUX0    ';'CTL0    ';'POLY1   ';'FILL1   ';'EPOCH1 '; ...
      'COUNT1 ';'MUX1    ';'CTL1    ';'POLY2   ';'FILL2   '; ...
      'EPOCH2 ';'COUNT2 ';'MUX2    ';'CTL2    ';'POLYALL'; ...
      'INITALL';'EPOCHAL';'COUNTAL';'MUXALL ';'CTLALL ';'MIXCODE'];
WHATINTSTR = ['CLK0    ';'COUNT0 ';'EPOCH0 ';'CODE0   ';'CLK1    ';...
      'COUNT1 ';'EPOCH1 ';'CODE1   ';'CLK2    ';'COUNT2 ';'EPOCH2 ';...
```

```
      'CODE2  ';'MIXCODE';'EXTRN0 ';'EXTRN1 ';'EXTRN2 '];
WHATLOADSTR = ['CODER0   ';'CODER1   ';'CODER2   ';'CODER_0_1';...
      'CODER_0_2';'CODER_1_2';'CODER_012'];
WHATCOLLECTSTR = ['CLK0';'CLK1';'CLK2'];
PLEDIT = 0;  % Has a value of 0 if not in edit mode
PLSAVED = 1;   % Has a value of 1 if the current list has been saved

a = figure('Color',[0.8 0.8 0.8], 'Units','points', ...
   'Colormap',mat0, 'Name','Process List Developer', ...
   'NumberTitle','off', 'PointerShapeCData',mat1, ...
   'Position',[50 30 386 385], 'Tag','pld',...
   'HandleVisibility','callback', 'CloseRequestFcn','pldclose');
LISTH = uicontrol('Parent',a, 'Units','points', ...
   'BackgroundColor',[1 1 1], 'FontName','Courier', ...
   'FontSize', 10, 'Position',[5 130 300 250], ...
   'String',[], 'Style','listbox', ...
   'Tag','processlistbox', 'Value',1, ...
   'Callback','plibupdate');
b = uicontrol('Parent',a, 'Units','points', ...
   'Position',[315 330 66 30], 'String','Edit', ...
   'Tag','Pushbutton1', 'Callback','pliedit');
b = uicontrol('Parent',a, 'Units','points', ...
   'Position',[315 284 66 30], 'String','Delete', ...
   'Tag','Pushbutton1', 'Callback','plidel');
b = uicontrol('Parent',a, 'Units','points', ...
   'Position',[315 238 66 30], 'String','Move Up', ...
   'Tag','Pushbutton1', 'Callback','plimove up');
b = uicontrol('Parent',a, 'Units','points', ...
   'Position',[315 192 66 30], 'String','Move Down', ...
   'Tag','Pushbutton1', 'Callback','plimove down');
b = uicontrol('Parent',a, 'Units','points', ...
   'Position',[315 146 66 30], 'String','Done', ...
   'Tag','Pushbutton1', 'Callback','plcomm');
WHENH = uicontrol('Parent',a, 'Units','points', ...
   'BackgroundColor',[1 1 1], 'Callback','plibupdate', ...
   'Position',[12.75 5 75 80], 'String',WHENSTR, ...
   'Style','listbox', 'Tag','when', 'Value',1);
ACTIONH = uicontrol('Parent',a, 'Units','points', ...
   'BackgroundColor',[1 1 1], 'Callback','plibupdate', ...
   'Position',[96.5 5 75 80], 'String',ACTIONSTR, ...
   'Style','listbox', 'Tag','action', 'Value',1);
WHATH = uicontrol('Parent',a, 'Units','points', ...
   'BackgroundColor',[1 1 1], 'Callback','plibupdate', ...
   'Position',[180.25 5 75 80], 'String',WHATSETSTR, ...
   'Style','listbox', 'Tag','what', 'Value',1);
VALUEH = uicontrol('Parent',a, 'Units','points', ...
   'BackgroundColor',[1 1 1], 'HorizontalAlignment','right', ...
   'Position',[263 67 81 18], 'String','0', ...
   'Style','edit', 'Tag','value');
BASEH = uicontrol('Parent',a, 'Units','points', ...
   'BackgroundColor',[0.8 0.8 0.8], 'HorizontalAlignment','left', ...
   'Position',[351 67 32 15], 'String','Octal', ...
   'Style','text', 'Tag','base');
b = uicontrol('Parent',a, 'Units','points', ...
   'BackgroundColor',[0.8 0.8 0.8], 'Position',[15.75 87 62 15], ...
   'String','When', 'Style','text', 'Tag','StaticText1');
b = uicontrol('Parent',a, 'Units','points', ...
```

```
   'BackgroundColor',[0.8 0.8 0.8], 'Position',[101.25 87 62 15], ...
   'String','Action', 'Style','text', 'Tag','StaticText1');
b = uicontrol('Parent',a, 'Units','points', ...
   'BackgroundColor',[0.8 0.8 0.8], 'Position',[187.5 87 62 15], ...
   'String','What', 'Style','text', 'Tag','StaticText1');
b = uicontrol('Parent',a, 'Units','points', ...
   'BackgroundColor',[0.8 0.8 0.8], 'Position',[273 87 62 15], ...
   'String','Value', 'Style','text', 'Tag','StaticText1');
b = uicontrol('Parent',a, 'Units','points', ...
   'BackgroundColor',[0.8 0.8 0.8], 'HorizontalAlignment','left', ...
   'Position',[351 87 45 15], 'String','Base', ...
   'Style','text', 'Tag','StaticText1');
b = uicontrol('Parent',a, 'Units','points', ...
   'Position',[270 35 70 30], 'String','Add', ...
   'Tag','AddPushbutton', 'Callback','plinew');
b = uicontrol('Parent', a, 'Units','points', 'Style', 'frame', ...
   'BackgroundColor',[0.8 0.8 0.8], 'Position',[1 125 403 1]);
b = uicontrol('Parent',a, 'Units','points', ...
   'BackgroundColor',[0.8 0.8 0.8], 'HorizontalAlignment','left', ...
   'Position',[15 102 200 20],'String','Process List Item Builder', ...
   'Style','text', 'Tag','StaticText1');

f = uimenu('Paren',a,'Label','List','Accelerator','L');
uimenu(f,'Label','Open','Callback','plopen','Accelerator','O');
uimenu(f,'Label','Save','Callback','plsave','Accelerator','S');
uimenu(f,'Label','Reset','Separator','on','Callback','plreset',...
   'Accelerator','R');
uimenu(f,'Label','Help','Separator','on','Callback','',...
   'Accelerator','H');

f = uimenu('Paren',a,'Label','Comment');
uimenu(f,'Label','View/Edit','Callback','plcommentedit');

PLUNDOH = uimenu('Paren',a,'Label','Undo');

if (nargin==1)  % Was a filename passed in?
   plopen(plfile);
else            % If not, set some variables.
   PLNAME = [];
   PLCOMMENT = [];
end
```

B.3 Process List Communicator GUI Code

The following MATLAB® script, "plcomm.m," creates the screen for the Process List Communicator developed for this research. The file "plcomm.mat" must also be present.

```
function plcomm()
% function plcomm()
%  Process List Communicator.  Allows the user to establish
%  communications with the box, download the user program,
%  execute the program, collect data, retrieve the data, and
%  export the data to the base workspace.
```

```
global PLDATABXS

load plcomm

plccheck.Units = 'points';
plccheck.BackgroundColor = [0.8 0.8 0.8];
plccheck.Style = 'checkbox';

plcedit.Units = 'points';
plcedit.BackgroundColor = [1 1 1];
plcedit.HorizontalAlignment = 'left';
plcedit.Style = 'edit';

plctext.Units = 'points';
plctext.BackgroundColor = [0.8 0.8 0.8];
plctext.HorizontalAlignment = 'left';
plctext.Style = 'text';

a = figure('Color',[0.8 0.8 0.8], 'Units','points', ...
   'Colormap',mat0, 'Name','Process List Communicator', ...
   'NumberTitle','off', 'PointerShapeCData',mat1, ...
   'Position',[200 100 380 288], 'Tag','plcomm',...
   'HandleVisibility','callback', 'CloseRequestFcn','plcommclose');
b = uicontrol('Parent',a, 'Units','points', ...
   'Callback','bootload', 'Enable','on','Position',[290 245 84 38], ...
   'String','Establish Comm', 'Tag','ecommbutton');
b = uicontrol('Parent',a, 'Units','points', ...
   'Callback','plcommunicator EXPORT', 'Enable','off',...
   'Position',[290 197 84 38], 'String','Export PL', ...
   'Tag','exportplbutton');
b = uicontrol('Parent',a, 'Units','points', ...
   'Callback','plcommunicator EXEC', 'Enable','off', ...
   'Position',[290 149 84 38], 'String','Execute PL', ...
   'Tag','execplbutton');
b = uicontrol('Parent',a, 'Units','points', ...
   'Callback','plcommunicator STOPCL', 'Enable','off', ...
   'Position',[290 101 84 38], 'String','Stop Collection', ...
   'Tag','stopcollbutton');
b = uicontrol('Parent',a, 'Units','points', ...
   'Callback','plcommunicator STARTCL', 'Enable','off', ...
   'Position',[190 101 84 38], 'String','Start Collection', ...
   'Tag','startcollbutton');
b = uicontrol('Parent',a, 'Units','points', ...
   'BackgroundColor',[1 1 1], 'HorizontalAlignment','right', ...
   'Position',[190 140 40 13], 'Style','edit', ...
   'Tag','collectamount');
b = uicontrol('Parent',a, 'Units','points', ...
   'BackgroundColor',[0.752941 0.752941 0.752941], ...
   'Position',[231 140 44 13], 'String',['CLK0';'CLK1';'CLK2'], ...
   'Style','popupmenu', 'Tag','collectclk', ...
   'Value',1);
b = uicontrol('Parent',a, plctext, 'Position',[190 155 85 13], ...
   'String','Number of Samples', 'Tag','numsamples');
b = uicontrol('Parent',a, 'Units','points', ...
   'Callback','plcommunicator STOPALL', 'Enable','off', ...
   'Position',[290 53 84 38], 'String','Stop All', ...
```

```
   'Tag','stopallbutton');
b = uicontrol('Parent',a, 'Units','points', ...
   'Callback','plcommunicator RETRIEVE', 'Enable','off', ...
   'Position',[290 5 84 38], 'String','Retrieve Data', ...
   'Tag','rdatabutton');
b = uicontrol('Parent',a, 'Units','points', ...
   'Callback','pldataextract', 'Enable','off', ...
   'Position',[160 20.4828 84 38], 'String','Export Data', ...
   'Tag','exdatabutton');
PLDATABXS(1,1) = uicontrol('Parent',a, plccheck, ...
   'Position',[8 230 50 13], 'String','CLK0', ...
   'Tag','clk0box');
PLDATABXS(2,1) = uicontrol('Parent',a, plccheck, ...
   'Position',[8 215 50 13], 'String','CODE0', ...
   'Tag','code0box');
PLDATABXS(3,1) = uicontrol('Parent',a, plccheck, ...
   'Position',[8 200 50 13], 'String','EPOCH0', ...
   'Tag','epoch0box');
PLDATABXS(4,1) = uicontrol('Parent',a, plccheck, ...
   'Position',[8 185 50 13], 'String','COUNT0', ...
   'Tag','count0box');
PLDATABXS(5,1) = uicontrol('Parent',a, plccheck, ...
   'Position',[8 170 50 13], 'String','CLK1', ...
   'Tag','clk1box');
PLDATABXS(6,1) = uicontrol('Parent',a, plccheck, ...
   'Position',[8 155 50 13], 'String','CODE1', ...
   'Tag','code1box');
PLDATABXS(7,1) = uicontrol('Parent',a, plccheck, ...
   'Position',[8 140 50 13], 'String','EPOCH1', ...
   'Tag','epoch1box');
PLDATABXS(8,1) = uicontrol('Parent',a, plccheck, ...
   'Position',[8 125 50 13], 'String','COUNT1', ...
   'Tag','count1box');
PLDATABXS(9,1) = uicontrol('Parent',a, plccheck, ...
   'Position',[8 110 50 13], 'String','CLK2', ...
   'Tag','clk2box');
PLDATABXS(10,1) = uicontrol('Parent',a, plccheck, ...
   'Position',[8 95 50 13], 'String','CODE2', ...
   'Tag','code2box');
PLDATABXS(11,1) = uicontrol('Parent',a, plccheck, ...
   'Position',[8 80 50 13], 'String','EPOCH2', ...
   'Tag','epoch2box');
PLDATABXS(12,1) = uicontrol('Parent',a, plccheck, ...
   'Position',[8 65 50 13], 'String','COUNT2', ...
   'Tag','count2box');
PLDATABXS(13,1) = uicontrol('Parent',a, plccheck, ...
   'Position',[8 50 50 13], 'String','MIXCODE', ...
   'Tag','mixcodebox');
PLDATABXS(14,1) = uicontrol('Parent',a, plccheck, ...
   'Position',[8 35 50 13], 'String','EXTRN0', ...
   'Tag','extrn0box');
PLDATABXS(15,1) = uicontrol('Parent',a, plccheck, ...
   'Position',[8 20 50 13], 'String','EXTRN1', ...
   'Tag','extrn1box');
PLDATABXS(16,1) = uicontrol('Parent',a, plccheck, ...
   'Position',[8 5 50 13], 'String','EXTRN2', ...
   'Tag','extrn2box');
```

```
PLDATABXS(1,2) = uicontrol('Parent',a, plcedit, ...
   'Position',[60 230 75 13], 'Tag','clk0vn');
PLDATABXS(2,2) = uicontrol('Parent',a, plcedit, ...
   'Position',[60 215 75 13], 'Tag','code0vn');
PLDATABXS(3,2) = uicontrol('Parent',a, plcedit, ...
   'Position',[60 200 75 13], 'Tag','epoch0vn');
PLDATABXS(4,2) = uicontrol('Parent',a, plcedit, ...
   'Position',[60 185 75 13], 'Tag','count0vn');
PLDATABXS(5,2) = uicontrol('Parent',a, plcedit, ...
   'Position',[60 170 75 13], 'Tag','clk1vn');
PLDATABXS(6,2) = uicontrol('Parent',a, plcedit, ...
   'Position',[60 155 75 13], 'Tag','code1vn');
PLDATABXS(7,2) = uicontrol('Parent',a, plcedit, ...
   'Position',[60 140 75 13], 'Tag','epoch1vn');
PLDATABXS(8,2) = uicontrol('Parent',a, plcedit, ...
   'Position',[60 125 75 13], 'Tag','count1vn');
PLDATABXS(9,2) = uicontrol('Parent',a, plcedit, ...
   'Position',[60 110 75 13], 'Tag','clk2vn');
PLDATABXS(10,2) = uicontrol('Parent',a, plcedit, ...
   'Position',[60 95 75 13], 'Tag','code2vn');
PLDATABXS(11,2) = uicontrol('Parent',a, plcedit, ...
   'Position',[60 80 75 13], 'Tag','epoch2vn');
PLDATABXS(12,2) = uicontrol('Parent',a, plcedit, ...
   'Position',[60 65 75 13], 'Tag','count2vn');
PLDATABXS(13,2) = uicontrol('Parent',a, plcedit, ...
   'Position',[60 50 75 13], 'Tag','mixcodevn');
PLDATABXS(14,2) = uicontrol('Parent',a, plcedit, ...
   'Position',[60 35 75 13], 'Tag','extrn0vn');
PLDATABXS(15,2) = uicontrol('Parent',a, plcedit, ...
   'Position',[60 20 75 13], 'Tag','extrn1vn');
PLDATABXS(16,2) = uicontrol('Parent',a, plcedit, ...
   'Position',[60 5 75 13], 'Tag','extrn2vn');
b = uicontrol('Parent',a, plctext, 'Position',[174 257 109 13], ...
   'String','Communication Established', ...
   'Tag','commest', 'Visible','off');
b = uicontrol('Parent',a, plctext, 'Position',[174 209 109 13], ...
   'String','PL Export Successful', ...
   'Tag','expsuccess', 'Visible','off');
b = uicontrol('Parent',a, plctext, 'Position',[174 85 109 13], ...
   'String','Data Collection Complete', ...
   'Tag','clctcomplete', 'Visible','off');
b = uicontrol('Parent',a, plctext, 'Position',[8 250 45 13], ...
    'String','Signal');
b = uicontrol('Parent',a, plctext, 'Position',[60 245 74 23], ...
    'String','Export to workspace as');
```

B.4 Process List Compiler Code

The following MATLAB® scripts, "plcompile.m" and "plcomplile2.m," compile the Process List into Siemens 80C166 machine language.

```
function [code, length] = plcompile
% function plcompile
```

```
%  Compiles the process list into machine code to be downloaded
%  to the microcontroller.  Retrieves the process list from the
%  Process List Developer.  Output is machine code for the Siemens
%  80C166 microcontroller.

global WHENH

whenstr = get(WHENH,'String');

% Sets the start address of the compiled Process List
baseaddress = hex2dec('05C0');
% Sets the jump address to a reti in the code
israddresses = ones(16,1).*1350;
% Maintains the isrCLK0 address unless changed later
israddresses(1) = 944;
% Maintains the isrCLK1 address unless changed later
israddresses(5) = 968;
% Maintains the isrCLK2 address unless changed later
israddresses(9) = 992;

% The code begins by resetting the STEL-1032
code = char([14 226 15 226]);
length = 4;

% Compile the INIT code
[newcode, newcl, data, dlength] = plcompile2('INIT');
if dlength ~= 0
   daddress = baseaddress + 32 + length + newcl + 4 + 2;
   daddressh = fix(daddress/256);
   daddressl = mod(daddress,256);
   newcode = [char([230 249 daddressl daddressh]) newcode];
   newcl = newcl + 4;
end
code = [code newcode];  % Add it to the current code
length = length + newcl;  % Add newcode length to current length
code = [code char([203 0])];  % Add ret
length = length+2;
code = [code data];
length = length + dlength;

isrcode = [];
% Go through all "When" events and compile their respective code
for i=2:17
   when = whenstr(i,:);
   [newcode, newcl, data, dlength] = plcompile2(when);
   if dlength ~= 0
      daddress = baseaddress + 32 + length + newcl + 4 + 6;
      daddressh = fix(daddress/256);
      daddressl = mod(daddress,256);
      newcode = [char([230 249 daddressl daddressh]) newcode];
      newcl = newcl + 4;
   end
   if newcl ~= 0
      if dlength ~= 0  % The ISR does write data to the STEL-1032
         % 32 is to account for the isrloc's
         israddresses(i-1)=baseaddress+length+32;
         isrcode = [isrcode char([236 249]),...
```

```
                newcode char([252 249 251 136]) data];
            length = length + newcl + 6 + dlength;
         else  % No data is not written to the STEL-1032
            % 32 is to account for the isrloc's
            israddresses(i-1)=baseaddress+length+32;
            isrcode = [isrcode newcode char([251 136])];
            length = length + newcl + 2;
         end
      end
   end
code = [code isrcode];

isrloc(2:2:32) = fix(israddresses./256);  % Condition the ISR addresses
isrloc(1:2:31) = mod(israddresses,256);   % to the proper form.
newcode = char(isrloc);
code = [newcode code];  % Put the ISR addresses at the beginning of the
length = length + 32;   % code.


function [code, length] = setintvector(whichone, isrlocation)
% Given which CAPCOM interrupt vector to set and the ISR location,
% returns the code to do so. isrlocation in decimal.
vectorloc = 64 + 4*(whichone-1);
isrloch = fix(isrlocation/256);
isrlocl = mod(isrlocation,256);
code = char([230 249 234 0 246 249 vectorloc 0 ...
      230 249 isrloch isrlocl 246 249 vectorloc+2 0]);
length = 16;



% New file begins here %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [code, length, data, dlength] = plcompile2(whichprocess)
% function plcompile2
%  Compiles the process list into machine code to be downloaded
%  to the microcontroller.  Retrieves the process list from the
%  Process List Developer.  Output is machine code for the Siemens
%  80C166 microcontroller.  Called by plcompile.

global ACTIONSTR WHATSETSTR WHATINTSTR WHATLOADSTR WHATCOLLECTSTR
global LISTH

% setaddresses contains the STEL-1032 register addresses,
% size of the register, and base of the Process List value
setaddresses = [0,4,8;4,4,8;8,4,8;12,4,10;16,1,10;17,1,2; ...
      32,4,8;36,4,8;40,4,8;44,4,10;48,1,10;49,1,2;...
      64,4,8;68,4,8;72,4,8;76,4,10;80,1,10;81,1,2;...
      96,4,8;100,4,8;104,4,8;108,4,10;112,1,10;113,1,2;128,1,2];

currentbox = get(LISTH,'String');   % Get the current Process List

code = []; data = [];
length = 0; dlength = 0;

% Find which lines to do in this pass
linestodo = strmatch(whichprocess,currentbox)';
for index = linestodo
```

```
% Get the current line from the list
currentline = currentbox(index,:);
% Parse the line into the tokens
[token, currentline]=strtok(currentline);
[token, currentline]=strtok(currentline);
action = strmatch(token,ACTIONSTR);
[token, currentline]=strtok(currentline);
switch action
case 1,  % Set
   what = strmatch(token,WHATSETSTR);
   value = strtok(currentline);
case 2,  % Load
   what = strmatch(token,WHATLOADSTR);
case {3,4},  % Enable/Disable Interrupt
   what = strmatch(token,WHATINTSTR);
case 5,  % Start Collect
   what = strmatch(token,WHATCOLLECTSTR);
   value = strtok(currentline);
case 6,  % Stop Collect
   what = strmatch(token,WHATCOLLECTSTR);
otherwise,
   disp(['Warning!!! Invalid Process List Item on line '...
         num2str(index)]);
end

j = (action-1)*50+what;
if (j>=1)&(j<=26)
   info = setaddresses(j,1:3);
   if info(2) == 4
      [newcode,newcl,newdata] = setregister4(j,info(1),...
         base2dec(value,info(3)));
      newdlength = 8;
   else
      [newcode,newcl,newdata] = setregister1(j,info(1),...
         base2dec(value,info(3)));
      newdlength = 2;
   end
   data = [data newdata];
   dlength = dlength + newdlength;
elseif (j>=51)&(j<=57)
   [newcode,newcl] = loadprn(j-50);
elseif (j>=101)&(j<=116)
   [newcode,newcl] = enableint(j-100);
elseif (j>=151)&(j<=162)
   [newcode,newcl] = disableint(j-150);
elseif (j>=201)&(j<=203)
   [newcode,newcl] = startcollect(j-200,str2num(value));
elseif (j>=251)&(j<=253)
   [newcode,newcl] = stopcollect(j-250);
else
   disp(j)
   disp(['Warning!!! Invalid Process List Item on line '...
         num2str(index)]);
   newcode = [];
   newcl = 0;
end
code = [code newcode];
```

```
   length = length + newcl;
end

function [code,length,dataout] = setregister4(setwhat, address, data)
% function [code,length] = setregister4(address, data)
%  Given a decimal address and data, returns the code for
%  setting the appropriate STEL-1032 four byte register.
%  length contains the length of the code.
data = dec2bin(data,32);
if (setwhat==1)|(setwhat==7)|(setwhat==13)|(setwhat==19)
%   if (1) %get(findobj('Tag','inversepoly'),'Value')==1)
%      temp = min(find(data=='1'));
%      data = [data(1:temp-1) data(32:-1:temp)];
%   end
   data1 = bin2dec(data(24:31));
   data2 = bin2dec(data(16:23));
   data3 = bin2dec(data(8:15));
   data4 = bin2dec(data(1:7));
else
   data1 = bin2dec(data(25:32));
   data2 = bin2dec(data(17:24));
   data3 = bin2dec(data(9:16));
   data4 = bin2dec(data(1:8));
end
code = char([232 57 232 57 232 57 232 57]);
dataout = char([address data1 address+1 data2...
      address+2 data3 address+3 data4]);
length = 8;

function [code,length,dataout] = setregister1(setwhat, address, data)
% function [code,length] = setregister1(address, data)
%  Given a decimal address and data, returns the code for
%  setting the appropriate STEL-1032 single byte register.
%  length contains the length of the code.
if (setwhat==5)|(setwhat==11)|(setwhat==17)|(setwhat==23)
   data = data - 1;  % Setting a MUX register; must account
end                  % for STEL-1032 offset.
code = char([232 57]);
dataout = char([address data]);
length = 2;

function [code,length] = loadprn(whichone)
%  Given which coder(s) to load, returns the code and code length
switch whichone
case 1,  % Load coder0
   code = char([30 226 31 226]);
   length = 4;
case 2,  % Load coder1
   code = char([46 226 47 226]);
   length = 4;
case 3,  % Load coder2
   code = char([62 226 63 226]);
   length = 4;
case 4,  % Load coder0 and 1
   code = char([102 226 249 255 118 226 6 0]);
   length = 8;
case 5,  % Load coder0 and 2
```

```
   code = char([102 226 245 255 118 226 10 0]);
   length = 8;
case 6,  % Load coder1 and 2
   code = char([102 226 243 255 118 226 12 0]);
   length = 8;
case 7,  % Load coder0, 1, and 2
   code = char([102 226 241 255 118 226 14 0]);
   length = 8;
end

function [code, length] = enableint(whichone)
%  Given which interrupt, returns the code to enable it
%  with a priority as given in the intpriority matrix.
%  NOTE:  This does not set the interrupt vector and
%         should not be done until the vector has been set!
%  Each enabled interrupt MUST have a different priority.
intpriority = [118,117,116,115,114,113,112,111,110,109,108,107,...
      106,105,104,103];
code = char([230 whichone+187 intpriority(whichone) 0]);
length = 4;

function [code, length] = disableint(whichone)
% Given which interrupt, returns the code to disable it
code = char([230 whichone+187 0 0]);
length = 4;

function [code, length] = stopcollect(whichone)
% Given which collection process to stop, returns code to disable it
[code, length] = disableint(4*whichone-3);

function [code, length] = startcollect(whichone, value)
% Given which collection process to start and how many samples
% to take, returns code to enable it
value = 2*value;
valueh = fix(value/256);
valuel = mod(value,256);
code = char([230 247 valuel valueh 230 248 whichone-1 0 202 0 30 5]);
length = 12;
```

B.5 Control Register Tool GUI Code

The following MATLAB® scripts, "setctltoolwindow.m" and "setctltool.m" are used as an aid to help set the STEL-1032's Control Register.

```
function setctltool(whattodo)
% function setctltool(whattodo)
%  Opens the Set Control Tool GUI.
%  Retrieves the values when user is done.

global VALUEH

switch whattodo
case 'create',
```

```
   setctltoolwindow;  % Load the GUI
   value=get(VALUEH,'String');
   if ((length(value)==8)&
       (bin2dec(value)>=0)&
       (bin2dec(value)<=255))

      set(findobj(gcf,'Tag','counteronepoch'),'Value', ...
          bin2dec(value(1:2))+1);
      set(findobj(gcf,'Tag','counteroncount'),'Value', ...
          bin2dec(value(3:4))+1);
      set(findobj(gcf,'Tag','fillonepoch'),'Value', ...
          bin2dec(value(5:6))+1);
      set(findobj(gcf,'Tag','filloncount'),'Value', ...
          bin2dec(value(7:8))+1);
   end

case 'finish',
   cntepoch = get(findobj(gcf,'Tag','counteronepoch'),'Value')-1;
   cntcnt = get(findobj(gcf,'Tag','counteroncount'),'Value')-1;
   epochepoch = get(findobj(gcf,'Tag','fillonepoch'),'Value')-1;
   epochcnt = get(findobj(gcf,'Tag','filloncount'),'Value')-1;
   teststring = [dec2bin(cntepoch,2) dec2bin(cntcnt,2) ...
         dec2bin(epochepoch,2) dec2bin(epochcnt,2)];

   close(gcf)
   set(VALUEH,'String',teststring);
end


% New file begins here %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function setctltoolwindow()
% function setctltoolwindow()
%  Helps the user the STEL-1032 Control Registers.

load setctltoolwindow

text.BackgroundColor = [0.8 0.8 0.8];
text.Style = 'text';
text.HorizontalAlignment = 'left';

popup.Style = 'popupmenu';
popup.BackgroundColor = [0.752941 0.752941 0.752941];

a = figure('Color',[0.8 0.8 0.8], 'Colormap',mat0, ...
   'Name','CTL Register Tool', 'NumberTitle','off', ...
   'PointerShapeCData',mat1, 'Position',[451 177 348 110], ...
   'HandleVisibility','callback', 'Tag','ctltool');
b = uicontrol('Parent',a, popup, 'Position',[35 5 90 24], ...
   'String',mat2, 'Tag','fillonepoch', 'Value',1);
b = uicontrol('Parent',a, text, 'Position',[9 33 150 17], ...
   'String','Reload PRN generator on');
b = uicontrol('Parent',a, popup, 'Position',[161 5 90 24], ...
   'String',mat3, 'Tag','filloncount', 'Value',1);
b = uicontrol('Parent',a, text, 'Position',[131 9.5 20 15], ...
   'String','or');
b = uicontrol('Parent',a, text, 'Position',[133 63 20 15], ...
   'String','or');
```

```
b = uicontrol('Parent',a, popup, 'Position',[159 58 90 24], ...
   'String',mat4, 'Tag','counteroncount', 'Value',1);
b = uicontrol('Parent',a, text, 'Position',[9 87 150 15], ...
   'String','Reload Counter on');
b = uicontrol('Parent',a, popup, 'Position',[36 58 90 24], ...
   'String',mat5, 'Tag','counteronepoch', 'Value',1);
b = uicontrol('Parent',a, 'Callback','setctltool finish', ...
   'Position',[258 34 73 37], 'String','OK', ...
   'Tag','OKPushbutton');
```

B.6 Other Functions

The following MATLAB® scripts are used by the previous scripts.

```
function bootload
% function bootload
%  Bootloads the Microcontroller and executes the BK software

global COMMH

load bootload

if isempty(COMMH)
   opencomm;
end

res = portcom('WRITE', COMMH, char(0));

incoming = portcom('READ',COMMH,1);

if ~isempty(incoming)
   if (incoming ~= 'U')
      disp('Communications Error!')
      h = warndlg('Communications Error!  Please press RESET and try
again.');
      return
   else
      disp('Communications Established.  Downloading software...')
   end
else
   disp('Communications Error!')
   h = warndlg('Communications Error!  Please press RESET and try
again.');
   return
end

res = portcom('WRITE',COMMH, boot1);
res = portcom('WRITE',COMMH, boot2);

incoming = portcom('READ',COMMH,'>');

res = portcom('WRITE', COMMH, 'D');
res = portcom('WRITE', COMMH, boot4);
```

```
incoming = portcom('READ',COMMH,char(10));
incoming = portcom('READ',COMMH,'>');

res = portcom('WRITE', COMMH, ['G' char(2) char(0)]);

incoming = portcom('READ',COMMH,char(10))
if (incoming == ' BK v1.0 Loaded')
   set(findobj('Tag','commest'),'Visible','on');
   set(findobj('Tag','ecommbutton'),'Enable','off');
   set(findobj('Tag','exportplbutton'),'Enable','on');
end



% New file begins here %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function opencomm
% function opencomm
%  Opens the Comm port and sets it to the specified
%  parameters.

global COMMH
COMMH = portcom('OPEN', 'COM2', 65000, 15000);

comstate.BaudRate = 19200;
comstate.ByteSize= 8;
comstate.StopBits= 0;

portcom('STATE',COMMH, comstate);



% New file begins here %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function plcommclose
% function plcommclose
%  Called when the PL Communicator window is closed.
%  Closes the serial port and clears the global variable
%  COMMH.

global COMMH

if ~isempty(COMMH)
   eval(['portcom(''CLOSE'',', int2str(COMMH),');'],'');
end

clear global COMMH PLDATABXS

delete(get(0,'CurrentFigure'))



% New file begins here %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function plcommentedit
% function plcommentedit
%  Called to allow the user to edit the PL Comment

global PLCOMMENT PLSAVED
```

```
prompt={'Enter a comment for this configuration'};
title='Comment';
lineNo=5;
temp=char(inputdlg(prompt,title,lineNo,{PLCOMMENT}));
if ~isempty(temp)
   PLCOMMENT=temp;
end

PLSAVED = 0;



% New file begins here %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function plcommunicator(whattodo)
% function plcommunicator(whattodo)
%  Bootloads the microcontroller and executes the BK software

global COMMH

if isempty(COMMH)
   disp('Communication has not been established!')
   return
end

switch whattodo
case 'EXPORT',
   exportPL;
case 'EXEC',
   executePL;
case 'STOPCL',
   stopCollect;
case 'STOPALL',
   stopAll;
case 'RETRIEVE',
   retrieve;
case 'STARTCL'
   startcollect;
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function exportPL
global COMMH PLLENGTH
set(findobj('Tag','expsuccess'),'Visible','off');
[code, length] = plcompile;
size = dec2hex(length,4);
res = portcom('WRITE',COMMH, 'X');
incoming = portcom('READ',COMMH,char(10));
if incoming ~= 'STPA'
   disp('Error trying to stop all!')
   disp('Error exporting Process List!')
   return
end
res = portcom('WRITE', COMMH, ['D' size]);
incoming = portcom('READ',COMMH,char(10));
if incoming ~= 'DOWN'
   disp('Export Error!')
```

```
   return
end
res = portcom('WRITE', COMMH, code);
incoming = portcom('READ',COMMH,char(10));
if (hex2dec(incoming)-hex2dec('05C0'))~=length
   disp('Error exporting Process List!')
else
   set(findobj('Tag','execplbutton'),'Enable','on');
   set(findobj('Tag','stopcollbutton'),'Enable','on');
   set(findobj('Tag','stopallbutton'),'Enable','on');
   set(findobj('Tag','rdatabutton'),'Enable','on');
   set(findobj('Tag','startcollbutton'),'Enable','on');
   set(findobj('Tag','expsuccess'),'Visible','on');
   PLLENGTH = length;
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function executePL
global COMMH
res = portcom('WRITE', COMMH, 'E');
incoming = portcom('READ',COMMH,char(10));
if incoming ~= 'EXEC'
   disp('Error trying to execute PL!')
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function stopCollect
global COMMH
res = portcom('WRITE',COMMH, 'F');
incoming = portcom('READ',COMMH,char(10));
if incoming ~= 'FINC'
   disp('Error trying to stop collection')
end
incoming = portcom('READ',COMMH,char(10));

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function stopAll
global COMMH
res = portcom('WRITE',COMMH, 'X');
incoming = portcom('READ',COMMH,char(10));
if incoming ~= 'STPA'
   disp('Error trying to stop all')
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function retrieve
global COMMH PLCDATA
res = portcom('WRITE',COMMH, 'S7918');
incoming = portcom('READ',COMMH,8);
size = hex2dec(incoming(5:8));
if (size==0)|(size>62000)
   disp('No data retrieved.')
else
   h = waitbar(0,'Retrieving data.  Please wait...');
   chunks = fix(size/1000);
   PLCDATA = [];
   if chunks ~= 0
```

```
      stepsize = 1/chunks;
      for i = 1:chunks
         incoming = portcom('READ',COMMH,1000);
         PLCDATA = [PLCDATA incoming];
         waitbar(stepsize*i);
      end
   end
   incoming = portcom('READ',COMMH,char(10));
   PLCDATA = [PLCDATA incoming];
   set(findobj('Tag','exdatabutton'),'Enable','on');
   close(h);
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function startcollect
global COMMH PLLENGTH
value = str2num(get(findobj('Tag','collectamount'),'String'));
whichclk = num2str(get(findobj('Tag','collectclk'),'Value')-1);
maxsamples = min([fix((62517-PLLENGTH)/2) 31000]);
if value > maxsamples
   disp('Number of samples has been reduced to the maximum allowable.')
   value = maxsamples;
   set(findobj('Tag','collectamount'),'String',num2str(value));
end
set(findobj('Tag','clctcomplete'),'Visible','off');
value = dec2hex(value,4);
res = portcom('WRITE', COMMH, ['C' value whichclk]);
incoming = portcom('READ',COMMH,char(10));
incoming = portcom('READ',COMMH,char(10))
set(findobj('Tag','clctcomplete'),'Visible','on');



% New file begins here %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function pldataextract
% function pldataextract
%  Extracts the various signals from the retrieved data.
%  Assigns them in the base workspace as the specified
%  variable.

global PLDATABXS PLCDATA

info=[1,1;8,1;4,1;2,1;16,1;128,1;64,1;32,1;1,2;8,2;4,2;2,2;...
      16,2;32,2;64,2;128,2];

for i=1:16
   if get(PLDATABXS(i,1),'Value')==1
      vn = get(PLDATABXS(i,2),'String');
      if isempty(vn)
         disp([get(PLDATABXS(i,1),'String')...
               ' not assigned.  No variable name given.'])
      else
         code =
min(1,bitand(info(i,1),double(PLCDATA(info(i,2):2:end))));
         assignin('base',vn,code);
      end
   end
```

```
end


% New file begins here %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function error = pldcheckvalue
% function error = pldcheckvalue
%  Checks the value currently in the Value edit box for range validity
global ACTIONH WHATH VALUEH

action = get(ACTIONH,'Value');
what = get(WHATH,'Value');
value = get(VALUEH,'String');
error = 0;

switch action
case 1,  % Action is Set
   switch what
   case {1,7,13,19},  % Poly, Octal data between 1 and 2^33-1
      x = base2dec(value,8);
      if ((x > 2^33-1)|(x < 1)|any(find(value > '7')))
         turnred;
         error = 1;
      else
         turnblack;
      end

   case {2,3,8,9,14,15,20,21},  % Octal data between 0 and 2^32-1
      x = base2dec(value,8);
      if ((x > 4294967295)|(x < 0)|any(find(value > '7')))
         turnred;
         error = 1;
      else
         turnblack;
      end

   case {4,10,16,22},  % Should be decimal data between 0 and 2^32-1
      x = str2num(value);
      if ((x > 4294967295)|(x < 0))
         turnred;
         error = 1;
      else
         turnblack;
      end

   case {5,11,17,23},  % Should be decimal data between 1 and 32
      x = str2num(value);
      if ((x > 32)|(x < 1))
         turnred;
         error = 1;
      else
         turnblack;
      end

   case {6,12,18,24,25},  % Should be binary data between 0 and 255
      x = bin2dec(value);
      if ((x > 255)|(x < 0))
```

```
         turnred;
         error = 1;
      else
         turnblack;
      end

   end
case 5,  % Action is Start Collect
   x = str2num(value);
   if ((x > 31000)|(x < 1))  % We can only collect up to about
      turnred;               % 31000 samples
      error = 1;
   else
      turnblack;
   end
end


function turnred
global VALUEH
set(VALUEH,'ForegroundColor','red');

function turnblack
global VALUEH
set(VALUEH,'ForegroundColor','black');



% New file begins here %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function pldclose
% function pldclose
%  Prompts the use to save the PL if not currently saved.
%  Closes window after response is processed.

global PLSAVED

if (PLSAVED==0)
   button = questdlg('The current process list has not been saved.  Are
you sure you want to exit?',...
      'Save before exit?','Save','Exit','Cancel','Cancel');
   if strcmp(button,'Save')
      plsave
   elseif strcmp(button,'Cancel')
      return
   end
end

clear global PLSAVED WHENSTR ACTIONSTR WHATSETSTR WHATINTSTR
clear global WHATLOADSTR WHATCOLLECTSTR LISTH WHENH ACTIONH BASEH
clear global WHATH VALUEH PLNAME PLCOMMENT PLEDIT PLUNDOH PLSAVED

delete(get(0,'CurrentFigure'))



% New file begins here %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function pliadd(newstring, targetindex)
```

```
% function pliadd(newstring, targetindex)
%  Adds a new Process List Item to the Process List.
%  If targetindex is not specified, the current list index is used.

global LISTH PLEDIT PLSAVED

oldstring = get(LISTH,'String');  % Get the current list
if (nargin<2)
   if (PLEDIT==1)  % Is this the result of an edit?
      index = get(LISTH,'Value');
      set(LISTH,'String',strvcat(oldstring(1:index-1,:),newstring, ...
         oldstring(index+1:end,:)));
      PLEDIT=0;
   else
      % Does this entry already exist?
      if any(strmatch(newstring(1:min(32,end)),oldstring))
         button = questdlg(...
      'A similar PLI already exists.  Do you want to add it again?',...
            'Add again?','Yes','No','No');
         if strcmp(button,'No')
            return
         end
      end
      if isempty(oldstring)
         dest = [];
      else
         dest = max(strmatch(newstring(1:6),oldstring(:,1:6)));
      end
      if isempty(dest)
         % Make the new Process List
         newstring = strvcat(oldstring,newstring);
         set(LISTH,'String',newstring, ...
            'Value', size(newstring,1));  % Establish the new list
      else
         set(LISTH,'String',strvcat(oldstring(1:dest,:),...
            newstring, oldstring(dest+1:end,:)),'Value',dest+1);
      end
   end
else
   set(LISTH,'String',strvcat(oldstring(1:targetindex-1,:),...
      newstring, oldstring(targetindex:end,:)));
end

PLSAVED = 0;



% New file begins here %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function plibupdate(when, action, what, value)
% function plibupdate(when, action, what, value)
%  Updates the PLIB values and lists.  If when, action,
%  what, and value are supplied then sets PLIB to those
%  values.  Otherwise, retrieves values from PLIB.

global WHENSTR ACTIONSTR WHATSETSTR WHATINTSTR WHATLOADSTR
global WHATCOLLECTSTR
global WHENH ACTIONH WHATH VALUEH BASEH PLEDIT
```

```
if (nargin>0)
   set(WHENH,'Value',when);
   set(ACTIONH,'Value',action);
   set(WHATH,'Value',what);
   set(VALUEH,'String',value);
else
   when = get(WHENH,'Value');  % Get the values of the PLIB objects
   action = get(ACTIONH,'Value');
   what = get(WHATH,'Value');
   value = get(VALUEH,'String');
end

PLEDIT = 0;
set(VALUEH,'ForegroundColor','black');
switch action
case 1,  % Set
   set(WHATH,'String',WHATSETSTR, 'Visible','on');

   if any(what==[6 12 18 24])
      set(VALUEH,'Visible','on','Enable','inactive', ...
         'ButtonDownFcn','setctltool create');
   else
      set(VALUEH,'Visible','on','Enable','on', ...
         'ButtonDownFcn','');
   end
case 2,  % Load
   if (get(WHATH,'Value')>7)
      set(WHATH,'Value',1);
   end
   set(WHATH,'String',WHATLOADSTR, 'Visible','on');
   set(VALUEH,'Visible','off');
case {3,4},  % Enable_Int and Disable_Int
   if (get(WHATH,'Value')>16)
      set(WHATH,'Value',1);
   end
   set(WHATH,'String',WHATINTSTR ,'Visible','on');
   set(VALUEH,'Visible','off');
case 5,  % Start_Collect
   if (get(WHATH,'Value')>3)
      set(WHATH,'Value',1);
   end
   set(WHATH,'String',WHATCOLLECTSTR,'Visible','on');
   set(BASEH,'String','Dec');
   set(VALUEH,'Visible','on','Enable','on');
case 6,  % Stop_Collect
   set(WHATH,'String',WHATCOLLECTSTR,'Visible','on');
   set(VALUEH,'Visible','off');
end

if action==1  % If Set is the action display correct base
   switch get(WHATH,'Value')
   case {1,2,3,7,8,9,13,14,15,19,20,21},
      set(BASEH,'String','Octal');
   case {4,5,10,11,16,17,22,23},
      set(BASEH,'String','Dec');
```

```
   case {6,12,18,24,25},
      set(BASEH,'String','Binary');
   end
end



% New file begins here %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function plidel(index)
% function plidel(index)
%  Deletes an Item from the Process List
%  If index is not supplied, uses current Value

global LISTH PLUNDOH PLEDIT PLSAVED

if (nargin==0)
   index=get(LISTH,'Value');
end

oldstring = get(LISTH,'String');   % Get the current list
if ~isempty(oldstring)
   delstring = oldstring(index,:);
   uimenu(PLUNDOH,'Label',delstring,'Callback', ...
      ['pliadd(''' delstring ''');' ...
         'delete(findobj(''Label'',''' delstring '''))']);
end
newstring = strvcat(oldstring(1:index-1,:),oldstring(index+1:end,:));
set(LISTH,'Value',min(max(1,size(newstring,1)),index));
set(LISTH,'String',newstring);     % Establish the new list

PLEDIT = 0;
PLSAVED = 0;



% New file begins here %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function pliedit(index)
% function pliedit(index)
%  Sets up the Process List Item Builder to edit the Process
%  List Item at index.  If index is not supplied, the current
%  Value is used.

global WHENSTR ACTIONSTR WHATSETSTR WHATINTSTR WHATLOADSTR
global WHATCOLLECTSTR
global LISTH PLEDIT

if (nargin==0)
   index=get(LISTH,'Value');
end

currentbox = get(LISTH,'String');   % Get the current list

currentline = currentbox(index,:);

[token, currentline]=strtok(currentline);

when = strmatch(token,WHENSTR);
```

```
[token, currentline]=strtok(currentline);
action = strmatch(token,ACTIONSTR);
[token, currentline]=strtok(currentline);
value=[];
switch action
case 1,
   what = strmatch(token,WHATSETSTR);
   value = strtok(currentline);
case 2,
   what = strmatch(token,WHATLOADSTR);
case {3,4},
   what = strmatch(token,WHATINTSTR);
case 5,
   what = strmatch(token,WHATCOLLECTSTR);
   value = strtok(currentline);
case 6,
   what = strmatch(token,WHATCOLLECTSTR);
end

plibupdate(when,action,what,value);
PLEDIT = 1;



% New file begins here %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function plimove(whichway)
% function plimove(whichway)
%  Moves the current PLI at index up or down one position,
%   if possible.

global LISTH PLSAVED PLEDIT

index=get(LISTH,'Value');

oldstring = get(LISTH,'String');   % Get the current list

switch (whichway)
case 'up',
   if (index==2)
      newstring = strvcat(oldstring(index,:), ...
         oldstring(index-1,:), oldstring(index+1:end,:));
      set(LISTH,'String',newstring);  % Establish the new list
   elseif (index>2)
     newstring = strvcat(oldstring(1:index-2,:), oldstring(index,:),...
         oldstring(index-1,:), oldstring(index+1:end,:));
     set(LISTH,'String',newstring);  % Establish the new list
   end
   % Move the selection bar up, if possible
   set(LISTH,'Value',max(1,index-1));

case 'down',
   maxdown = size(oldstring,1);  % How far down can it go

   if (index<(maxdown-1))
      newstring=strvcat(oldstring(1:index-1,:),oldstring(index+1,:),...
         oldstring(index,:), oldstring(index+2:end,:));
      set(LISTH,'String',newstring);  % Establish the new list
```

```
   elseif (index==(maxdown-1))
      newstring=strvcat(oldstring(1:index-1,:),oldstring(index+1,:),...
         oldstring(index,:));
      set(LISTH,'String',newstring);  % Establish the new list
   end
   % Move the selection bar down, if possible
   set(LISTH,'Value',min(maxdown,index+1));
end

PLSAVED = 0;
PLEDIT = 0;



% New file begins here %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function plinew
% function plinew
%  Constructs the new Process List Item string by reading
%  the values of the Process List Item Builder and passes
%  it to addpli.  Checks the values validity and aborts if
%  it is invalid.

error = pldcheckvalue;
if (error == 1)
   return
end

global LISTH WHENH ACTIONH WHATH VALUEH BASEH

when = get(WHENH,'Value');  % Get PLIB object values
action = get(ACTIONH,'Value');
what = get(WHATH,'Value');
value = get(VALUEH,'String');

whenstr = get(WHENH,'String');  % Get the current list box strings
actionstr = get(ACTIONH,'String');
whatstr = get(WHATH,'String');
base = get(BASEH,'String');

switch action
case {1,5}
   newstring = sprintf('%-7s %-13s %-8s %11s %-6s',whenstr(when,:), ...
      actionstr(action,:), whatstr(what,:), value, base);
case {2,3,4,6},
   newstring = sprintf('%-7s %-13s %-8s',whenstr(when,:), ...
      actionstr(action,:), whatstr(what,:));
end

pliadd(newstring);  % Put the new PLI in the list



% New file begins here %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function plopen(filename)
% function plopen(filename)
%  Reads a Process List from file filename.
%  If filename is not supplied, a dialog box prompts for it.
```

```
global LISTH PLNAME PLCOMMENT PLSAVED

if (nargin==0)
   [getplffile, newpath] = uigetfile('*.plf', 'Open');
   if (getplffile==0)  % Error occured or user canceled the action
      return
   end
   PLNAME = [newpath getplffile];
else
   PLNAME = filename;
end


fid = fopen(PLNAME,'r');

incoming = fscanf(fid,'%c');

fclose(fid);

set(LISTH,'String',[],'Value',1);  % Reset the Process List%

[temp, incoming] = strtok(incoming,char(10));
[temp, incoming] = strtok(incoming,char(10));
PLCOMMENT = temp(3:end);

newlist = [];
while ~isempty(incoming)
   [partlist, incoming] = strtok(incoming,char(10));
   newlist = strvcat(newlist,partlist);
end

pliadd(newlist);
set(gcf,'Name',['Process List Developer - ' PLNAME]);

PLSAVED = 1;



% New file begins here %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function plqb2pld
% function plqb2pld
%  Reads the values from the Quick Builder GUI and transfers
%  them to the Process List Developer as a Process List

[errors,data] = plqbhelper('validate');
if (errors>0)
   return
end

values{1} = data.poly0;
values{2} = data.fill0;
values{3} = data.epoch0;
values{4} = data.counter0;
values{5} = data.mux0;
counte    = data.counte0;
countc    = data.countc0;
```

```
fille     = data.fille0;
fillc     = data.fillc0;
values{6} =[dec2bin(counte-1,2) dec2bin(countc-1,2) ...
      dec2bin(fille-1,2) dec2bin(fillc-1,2)];
what = ['POLY0  ';'FILL0  ';'EPOCH0 ';'COUNT0 '; ...
      'MUX0   ';'CTL0   '];
base = ['Octal ';'Octal ';'Octal ';'Dec   ';'Dec   ';'Binary'];

if (get(findobj('Tag','coder1'),'Value')==1)
   x = size(values,2);
   values{x+1} = data.poly1;
   values{x+2} = data.fill1;
   values{x+3} = data.epoch1;
   values{x+4} = data.counter1;
   values{x+5} = data.mux1;
   counte      = data.counte1;
   countc      = data.countc1;
   fille       = data.fille1;
   fillc       = data.fillc1;
   values{x+6} =[dec2bin(counte-1,2) dec2bin(countc-1,2) ...
         dec2bin(fille-1,2) dec2bin(fillc-1,2)];
   what = [what;'POLY1  ';'FILL1  ';'EPOCH1 ';'COUNT1 '; ...
         'MUX1   ';'CTL1   '];
   base = [base;'Octal ';'Octal ';'Octal ';'Dec   '; ...
         'Dec   ';'Binary'];
end

if (get(findobj('Tag','coder2'),'Value')==1)
   x = size(values,2);
   values{x+1} = data.poly2;
   values{x+2} = data.fill2;
   values{x+3} = data.epoch2;
   values{x+4} = data.counter2;
   values{x+5} = data.mux2;
   counte      = data.counte2;
   countc      = data.countc2;
   fille       = data.fille2;
   fillc       = data.fillc2;
   values{x+6} =[dec2bin(counte-1,2) dec2bin(countc-1,2) ...
         dec2bin(fille-1,2) dec2bin(fillc-1,2)];
   what = [what;'POLY2  ';'FILL2  ';'EPOCH2 ';'COUNT2 '; ...
         'MUX2   ';'CTL2   '];
   base = [base;'Octal ';'Octal ';'Octal ';'Dec   '; ...
         'Dec   ';'Binary'];
end

if (get(findobj('Tag','mixcode'),'Value')==1)
   x = size(values,2);
   values{x+1} = [data.combiner7...
          data.combiner6...
          data.combiner5...
          data.combiner4...
          data.combiner3...
          data.combiner2...
          data.combiner1...
          data.combiner0];
   what = [what;'MIXCODE'];
```

```
   base = [base;'Binary'];
end

if isempty(findobj('Tag','pld'))
   pld
end

for i=1:size(what,1)
   newstring = sprintf('%-7s %-13s %-8s %11s %-6s','INIT', ...
      'Set', what(i,:) , char(values(i)), base(i,:));
   pliadd(newstring);
end



% New file begins here %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [output,data] = plqbhelper(whattodo)
% function plqbhelper(whattodo)
%  Process List Quick Builder Helper functions.
%  Validates values, saves, and opens setup screens.
switch whattodo,
case 'validate'
   [output,data] = goValidate;
case 'save'
   goSave;
case 'open'
   goOpen;
end

function [errors,data] = goValidate
errors = 0;
data = readplqbvalues;
if checkOctalPoly(data.poly0)
   set(findobj('Tag','poly0text'),'ForegroundColor','red');
   errors = 1;
else
   set(findobj('Tag','poly0text'),'ForegroundColor','black')
end
if checkOctalPoly(data.poly1)
   set(findobj('Tag','poly1text'),'ForegroundColor','red');
   errors = 1;
else
   set(findobj('Tag','poly1text'),'ForegroundColor','black');
end
if checkOctalPoly(data.poly2)
   set(findobj('Tag','poly2text'),'ForegroundColor','red');
   errors = 1;
else
   set(findobj('Tag','poly2text'),'ForegroundColor','black');
end
if checkOctalFill(data.fill0)
   set(findobj('Tag','fill0text'),'ForegroundColor','red');
   errors = 1;
else
   set(findobj('Tag','fill0text'),'ForegroundColor','black');
end
if checkOctalFill(data.fill1)
```

```
   set(findobj('Tag','fill1text'),'ForegroundColor','red');
   errors = 1;
else
   set(findobj('Tag','fill1text'),'ForegroundColor','black');
end
if checkOctalFill(data.fill2)
   set(findobj('Tag','fill2text'),'ForegroundColor','red');
   errors = 1;
else
   set(findobj('Tag','fill2text'),'ForegroundColor','black');
end
if checkOctalEpoch(data.epoch0)
   set(findobj('Tag','epoch0text'),'ForegroundColor','red');
   errors = 1;
else
   set(findobj('Tag','epoch0text'),'ForegroundColor','black');
end
if checkOctalEpoch(data.epoch1)
   set(findobj('Tag','epoch1text'),'ForegroundColor','red');
   errors = 1;
else
   set(findobj('Tag','epoch1text'),'ForegroundColor','black');
end
if checkOctalEpoch(data.epoch2)
   set(findobj('Tag','epoch2text'),'ForegroundColor','red');
   errors = 1;
else
   set(findobj('Tag','epoch2text'),'ForegroundColor','black');
end
if checkDecCount(data.counter0)
   set(findobj('Tag','counter0text'),'ForegroundColor','red');
   errors = 1;
else
   set(findobj('Tag','counter0text'),'ForegroundColor','black');
end
if checkDecCount(data.counter1)
   set(findobj('Tag','counter1text'),'ForegroundColor','red');
   errors = 1;
else
   set(findobj('Tag','counter1text'),'ForegroundColor','black');
end
if checkDecCount(data.counter2)
   set(findobj('Tag','counter2text'),'ForegroundColor','red');
   errors = 1;
else
   set(findobj('Tag','counter2text'),'ForegroundColor','black');
end
if checkDecMux(data.mux0)
   set(findobj('Tag','phase_mux0text'),'ForegroundColor','red');
   errors = 1;
else
   set(findobj('Tag','phase_mux0text'),'ForegroundColor','black');
end
if checkDecMux(data.mux1)
   set(findobj('Tag','phase_mux1text'),'ForegroundColor','red');
   errors = 1;
else
```

```
   set(findobj('Tag','phase_mux1text'),'ForegroundColor','black');
end
if checkDecMux(data.mux2)
   set(findobj('Tag','phase_mux2text'),'ForegroundColor','red');
   errors = 1;
else
   set(findobj('Tag','phase_mux2text'),'ForegroundColor','black');
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function result = checkOctalFill(data)
result = ((base2dec(data,8)<1)|(base2dec(data,8)>(2^32-1))| ...
      any(find(data > '7')));
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function result = checkOctalPoly(data)
result = ((base2dec(data,8)<1)|(base2dec(data,8)>(2^33-1))| ...
      any(find(data > '7')));
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function result = checkOctalEpoch(data)
result = ((base2dec(data,8)>(2^32-1))|any(find(data > '7')));
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function result = checkDecCount(data)
result = ((str2num(data)<0)|(str2num(data)>(2^32-1)));
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function result = checkDecMux(data)
result = ((str2num(data)<1)|(str2num(data)>32));
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function goSave
[newqbffile, newpath] = uiputfile('*.qbf.mat', 'Save As');
if newqbffile ~= 0
   data = readplqbvalues;
   eval(['save ', newpath, newqbffile, ' data']);
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function goOpen
[qbffile, qbfpath] = uigetfile('*.qbf.mat', 'Load');
if qbffile ~= 0
   eval(['load ', qbfpath, qbffile]);
   setplqbvalues(data);
end



% New file begins here %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function plreset
% function plreset
%  Resets the Process List.
global LISTH PLNAME PLCOMMENT PLSAVED

button = questdlg('Are you sure you want to reset the Process
List?',...
   'Reset?','Yes','Cancel','Cancel');
if strcmp(button,'Yes')
   set(LISTH,'String',[],'Value',1);
   PLNAME = [];
   PLCOMMENT = [];
   set(gcf,'Name','Process List Developer');
```

```
   PLSAVED = 1;
end



% New file begins here %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function plsave
% function plsave
%  Saves the Process List
global LISTH PLNAME PLCOMMENT PLSAVED

if isempty(PLNAME)
   PLNAME = '*.plf';
end

[newplffile, newpath] = uiputfile(PLNAME, 'Save As');

if (newplffile==0)  % Error occured or user canceled the action
   return
end

PLNAME = [newpath, newplffile];

pl = get(LISTH,'String');

prompt={'Enter a comment for this configuration'};
title='Comment';
lineNo=5;
temp=char(inputdlg(prompt,title,lineNo,{PLCOMMENT}));
if ~isempty(temp)
   PLCOMMENT=temp;
end

fid = fopen(PLNAME,'w');

fprintf(fid,'%% Process List File created by PLD v0.5\n');
fprintf(fid, '%% %s\n',PLCOMMENT');
for i=1:size(pl,1)
   fprintf(fid,'%s\n',pl(i,:));
end

fclose(fid);

set(gcf,'Name',['Process List Developer - ' PLNAME]);

PLSAVED = 1;



% New file begins here %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function info = readplqbvalues
% Reads the values from the Process List Quick Builder GUI

info.poly0    =get(findobj('Tag','poly0'),'String');
info.fill0    =get(findobj('Tag','fill0'),'String');
info.epoch0   =get(findobj('Tag','epoch0'),'String');
info.counter0 =get(findobj('Tag','counter0'),'String');
```

```
info.mux0      =get(findobj('Tag','phase_mux0'),'String');
info.counte0   =get(findobj('Tag','counter0epoch'),'Value');
info.countc0   =get(findobj('Tag','counter0count'),'Value');
info.fille0    =get(findobj('Tag','fill0epoch'),'Value');
info.fillc0    =get(findobj('Tag','fill0count'),'Value');

info.coder1chk=get(findobj('Tag','coder1'),'Value');
info.poly1     =get(findobj('Tag','poly1'),'String');
info.fill1     =get(findobj('Tag','fill1'),'String');
info.epoch1    =get(findobj('Tag','epoch1'),'String');
info.counter1  =get(findobj('Tag','counter1'),'String');
info.mux1      =get(findobj('Tag','phase_mux1'),'String');
info.counte1   =get(findobj('Tag','counter1epoch'),'Value');
info.countc1   =get(findobj('Tag','counter1count'),'Value');
info.fille1    =get(findobj('Tag','fill1epoch'),'Value');
info.fillc1    =get(findobj('Tag','fill1count'),'Value');

info.coder2chk=get(findobj('Tag','coder2'),'Value');
info.poly2     =get(findobj('Tag','poly2'),'String');
info.fill2     =get(findobj('Tag','fill2'),'String');
info.epoch2    =get(findobj('Tag','epoch2'),'String');
info.counter2  =get(findobj('Tag','counter2'),'String');
info.mux2      =get(findobj('Tag','phase_mux2'),'String');
info.counte2   =get(findobj('Tag','counter2epoch'),'Value');
info.countc2   =get(findobj('Tag','counter2count'),'Value');
info.fille2    =get(findobj('Tag','fill2epoch'),'Value');
info.fillc2    =get(findobj('Tag','fill2count'),'Value');

info.mixcodechk=get(findobj('Tag','mixcode'),'Value');
info.combiner7 = get(findobj('Tag','combiner7'),'String');
info.combiner6 = get(findobj('Tag','combiner6'),'String');
info.combiner5 = get(findobj('Tag','combiner5'),'String');
info.combiner4 = get(findobj('Tag','combiner4'),'String');
info.combiner3 = get(findobj('Tag','combiner3'),'String');
info.combiner2 = get(findobj('Tag','combiner2'),'String');
info.combiner1 = get(findobj('Tag','combiner1'),'String');
info.combiner0 = get(findobj('Tag','combiner0'),'String');



% New file begins here %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function setplqbvalues(info)
% Sets all of the values on the Process List Quick Builder GUI
set(findobj('Tag','poly0'),'String',info.poly0);
set(findobj('Tag','fill0'),'String',info.fill0);
set(findobj('Tag','epoch0'),'String',info.epoch0);
set(findobj('Tag','counter0'),'String',info.counter0);
set(findobj('Tag','phase_mux0'),'String',info.mux0);
set(findobj('Tag','counter0epoch'),'Value',info.counte0);
set(findobj('Tag','counter0count'),'Value',info.countc0);
set(findobj('Tag','fill0epoch'),'Value',info.fille0);
set(findobj('Tag','fill0count'),'Value',info.fillc0);

set(findobj('Tag','coder1'),'Value',info.coder1chk);
set(findobj('Tag','poly1'),'String',info.poly1);
set(findobj('Tag','fill1'),'String',info.fill1);
set(findobj('Tag','epoch1'),'String',info.epoch1);
```

```
set(findobj('Tag','counter1'),'String',info.counter1);
set(findobj('Tag','phase_mux1'),'String',info.mux1);
set(findobj('Tag','counter1epoch'),'Value',info.counte1);
set(findobj('Tag','counter1count'),'Value',info.countc1);
set(findobj('Tag','fill1epoch'),'Value',info.fille1);
set(findobj('Tag','fill1count'),'Value',info.fillc1);

set(findobj('Tag','coder2'),'Value',info.coder2chk);
set(findobj('Tag','poly2'),'String',info.poly2);
set(findobj('Tag','fill2'),'String',info.fill2);
set(findobj('Tag','epoch2'),'String',info.epoch2);
set(findobj('Tag','counter2'),'String',info.counter2);
set(findobj('Tag','phase_mux2'),'String',info.mux2);
set(findobj('Tag','counter2epoch'),'Value',info.counte2);
set(findobj('Tag','counter2count'),'Value',info.countc2);
set(findobj('Tag','fill2epoch'),'Value',info.fille2);
set(findobj('Tag','fill2count'),'Value',info.fillc2);

set(findobj('Tag','mixcode'),'Value',info.mixcodechk);
set(findobj('Tag','combiner7'),'String',info.combiner7);
set(findobj('Tag','combiner6'),'String',info.combiner6);
set(findobj('Tag','combiner5'),'String',info.combiner5);
set(findobj('Tag','combiner4'),'String',info.combiner4);
set(findobj('Tag','combiner3'),'String',info.combiner3);
set(findobj('Tag','combiner2'),'String',info.combiner2);
set(findobj('Tag','combiner1'),'String',info.combiner1);
set(findobj('Tag','combiner0'),'String',info.combiner0);
```

APPENDIX C

MICROCONTROLLER CODE

This Appendix contains microcontroller "resident" code in Siemens 80C166 assembly language.

This code was assembled using Rigel Corporation's Reads 166 Compiler/Assembler 3.00.

```
collectCount equ R5
collectStop  equ R4
; --- SFRs ---
     MDL    defr 0FE0Eh
     MDH    defr 0FE0Ch
     DP3    defr 0FFC6h
     S0CON  defr 0FFB0h
     S0BG   defr 0FEB4h
     S0RBUF defr 0FEB2h
     S0TBUF defr 0FEB0h
     S0TIC  defr 0FF6Ch
     S0RIC  defr 0FF6Eh
     S0EIC  defr 0FF70h
     ZEROS  defr 0FF1Ch
     ONES   defr 0FF1Eh
; --- bits ---
     S0TIR  defb S0TIC.7
     S0RIR  defb S0RIC.7
; ------------------------------------
; -- reset and hardware trap vectors --
; ------------------------------------
     org   0
     jmpa  cc_UC, _startup
     org   28h
     jmpa  cc_UC, _startup
; ------------------------------------
; --- startup code ---
; ------------------------------------
     org   00200h
_startup:
     mov   SYSCON, #0008Dh
     nop
     mov   SYSCON, #004CDh
     nop
     jmps  0, next1
next1:
     mov   DPP0, #0
     mov   DPP1, #1
     mov   DPP2, #2
     mov   DPP3, #3
     mov   PSW, ZEROS
     bclr  NMI
     mov   CP, #0FC00h
     mov   SP, #0FC00h
     mov   STKOV, #0FA00h
```

```
     mov   STKUN, #0FC00h
     bset  P3.13
     bset  DP3.13
     bset  DP3.10
     mov   S0CON, #8011h
     mov   S0BG, #1Fh
     mov   S0TIC, #0
     mov   S0RIC, #0
     mov   S0EIC, #0
     DISWDT
     EINIT

; --- insert your startup code here ---

     calla cc_UC, main
; --- end of startup ------------------
main :
     mov   DP2, #00000h
     mov   DP3, #024FFh
     mov   P3, #0EFFFh
     mov   R0, #02222h
     mov   CCM0, R0
     mov   CCM1, R0
     mov   CCM2, R0
     mov   CCM3, R0
     bclr  P3_0
     bset  P3_0
     bset  IEN
     mov   R3, #0F9FEh  ; R3 contains the output address
     mov    R6, #0FFC0h ; R6 contains the P2 address
     mov    R1, #plist2
     add    R1, #2
     mov    collectStart, R1
     mov   collectCount, R1
     mov   R1, #000CBh
     mov   plist2, R1
     calla cc_UC, id

_0023 :
     calla cc_UC, waitGetChar
     cmp   R0, #49h           ;'I'
     jmpr  cc_EQ, _0025
     cmp   R0, #053h          ;'S'
     jmpr  cc_EQ, _0026
     cmp   R0, #058h          ;'X'
     jmpr  cc_EQ, _0027
     cmp   R0, #045h          ;'E'
     jmpr  cc_EQ, _0028
     cmp   R0, #043h          ;'C'
     jmpr  cc_EQ, _0029
     cmp   R0, #044h          ;'D'
     jmpr  cc_EQ, _0030
     cmp   R0, #046h          ;'F'
     jmpr  cc_EQ, _0031
     cmp   R0, #052h          ;'R'
     jmpr  cc_EQ, _0032
```

```
     jmpr  cc_UC, _0033

_0025 :
     calla cc_UC, id
     jmpr  cc_UC, _0023
_0026 :
     calla cc_UC, getHexInt
     calla cc_UC, send
     jmpr  cc_UC, _0023
_0027 :
     calla cc_UC, stopAll
     jmpr  cc_UC, _0023
_0028 :
     calla cc_UC, execute
     jmpr  cc_UC, _0023
_0029 :
     calla cc_UC, getHexInt
     calla cc_UC, collectData
     jmpr  cc_UC, _0023
_0030 :
     calla cc_UC, getHexInt
     calla cc_UC, download
     jmpr  cc_UC, _0023
_0031 :
     calla cc_UC, stopCollect
     jmpr  cc_UC, _0023
_0032 :
     calla cc_UC, reset
     jmpr  cc_UC, _0023
_0033 :
     mov   R0, #S_0000
     calla cc_UC, SendStr

     jmpr  cc_UC, _0023
; --- main ends ------------------------

SendHexInt :      ; Input should be in R0
     push   R1
     push   R0
     push   R0
     push   R0
     push   R0
     mov    R1, #0Ch
_0062 :
     shr    R0, R1
     and    R0, #0Fh
     cmp    R0, #0Ah
     jmpr   cc_SLT, _0061
     add    R0, #00007h
_0061 :
     add    R0, #00030h
     calla  cc_UC, SendChar
     pop    R0
     sub    R1, #4
     cmp    R1, #0
     jmpr   cc_SGE, _0062
```

```
     pop    R1    ; Note: R0 has already been popped
     ret
;----- End SendHexInt -----
SendChar :  ; Input should be in R0
     movb   RH0, #0
     mov    S0TBUF, R0
_0075 :
     jnb    S0TIR, _0075
     mov    S0TIC, #00000h
     ret
;----- End SendChar -----
SendStr :   ; Input (add of string) should be in R0
      push  R1
      push  R0
      mov   R1, R0
_0080 :
     movb   RL0, [R1+]
     cmp    R0, ZEROS
     jmpr   cc_EQ, _0081
     calla  cc_UC, SendChar
     jmpr   cc_UC, _0080
_0081 :
      pop   R0
      pop   R1
      ret
;----- End SendStr -----
getHexInt : ; Output is in R0
      push  R2
      push  R1
      mov   R2, #0
      mov   R1, #0Ch
_0089 :
      calla cc_UC, waitGetChar
      cmp   R0, #65
      jmpr  cc_ULT, _0082
      sub   R0, #7
_0082 :
      sub   R0, #48
      shl   R0, R1
      add   R2, R0
      sub   R1, #4
      cmp   R1, #0
      jmpr  cc_SGE, _0089
      mov   R0, R2
      pop   R1
      pop   R2
      ret
;----- End getHexInt -----
waitGetChar :     ; Output is in R0
_0091 :
      jnb   S0RIR, _0091
      bclr  S0RIR
      mov   S0RIC, #00000h
      mov   R0, S0RBUF
      ret
;----- End waitGetChar -----
; --- insert interrupt vector ---
```

```
      pushlc
      org   0040h
      jmpa  cc_UC, isrCLK0
      poplc
isrCLK0 :
      mov   [collectCount+], [R6]
      cmp   collectCount, collectStop
      jmpr  cc_UGT, _0098
      reti
_0098 :
      mov   CC0IC, #00000h
      mov   R0, #S_0001
      calla cc_UC, SendStr
      sub   collectCount, #2
      reti
;----- End isrCLK0 -----
; --- insert interrupt vector ---
      pushlc
      org   0050h
     jmpa   cc_UC, isrCLK1
     poplc
isrCLK1 :
      mov   [collectCount+], [R6]
      cmp   collectCount, collectStop
      jmpr  cc_UGT, _0102
      reti
_0102 :
      mov   CC4IC, #00000h
      mov   R0, #S_0001
      calla cc_UC, SendStr
      sub    collectCount, #2
      reti
;----- End isrCLK1 -----
; --- insert interrupt vector ---
      pushlc
      org   0060h
      jmpa  cc_UC, isrCLK2
      poplc
isrCLK2 :
      mov   [collectCount+], [R6]
      cmp   collectCount, collectStop
      jmpr  cc_UGT, _0106
      reti
_0106 :
      mov   CC8IC, #00000h
      mov   R0, #S_0001
      calla cc_UC, SendStr
      sub   collectCount, #2
      reti
;----- End isrCLK2 -----
send :      ; Input (# of samples to send) is in R0
      push  R0
      mov   R0, #S_0004
      calla cc_UC, SendStr
      pop   R1
      shl   R1, #1
      mov   R2, collectStart
```

```
      add   R1, R2
      cmp   R1, collectCount
      jmpr  cc_ULT, _0110
      mov   R1, collectCount
_0110 :
      mov   R0, R1
      sub   R0, R2
      calla cc_UC, SendHexInt
_0108 :
      cmp   R2, R1
      jmpr  cc_UGE, _0109
      movb  RL0, [R2+]
      calla cc_UC, SendChar
      jmpr  cc_UC, _0108
_0109 :
      mov   R0, #00Ah
      calla cc_UC, SendChar
      ret
;----- End send -----
reset :
      calla cc_UC, stopAll
      mov   R0, #plist2
      add   R0, #2
      mov   collectStart, R0
      mov   collectCount, R0
      mov   collectStop, R0
      mov   R0, #000CBh
      mov   plist2, R0
      mov   R0, #S_0007
      calla cc_UC, SendStr
      ret
;----- End reset -----
stopAll :
      mov   R0, #00000h
      mov   CC0IC, R0
      mov   CC1IC, R0
      mov   CC2IC, R0
      mov   CC3IC, R0
      mov   CC4IC, R0
      mov   CC5IC, R0
      mov   CC6IC, R0
      mov   CC7IC, R0
      mov   CC8IC, R0
      mov   CC9IC, R0
      mov   CC10IC, R0
      mov   CC11IC, R0
      mov   CC12IC, R0
      mov   CC13IC, R0
      mov   CC14IC, R0
      mov   CC15IC, R0
      mov   R0, #S_0005
      calla cc_UC, SendStr
      ret
;----- End stopAll -----
stopCollect :
      mov   R0, #S_0006
      calla cc_UC, SendStr
```

```
      mov   collectStop, collectCount
      mov   R0, collectCount
      mov   R1, collectStart
      sub   R0, R1
      calla cc_UC, SendHexInt
      mov   R0, #0000Ah
      calla cc_UC, SendChar
      ret
;----- End stopCollect -----
download :  ; # bytes to get comes in R0
      push  R0
      mov   R0, #S_0008
      calla cc_UC, SendStr
      pop   R1
      mov   R2, #plist
      add   R1, R2
_0116 :
      cmp   R2, R1
      jmpr  cc_UGE, _0117
      calla cc_UC, waitGetChar
      movb  [R2], RL0
      add   R2, #1
      jmpr  cc_UC, _0116
_0117 :
      add   R2, #1      ; Need to insure that
      and   R2, #0FFFEh ; collecStart is even
      mov   collectStart, R2
      mov   R0, R2
      calla cc_UC, SendHexInt
      mov   R0, #0000Ah
      calla cc_UC, SendChar
      ret
;----- End download -----
execute :
      mov   R0, #S_0009
      calla cc_UC, SendStr
      jmpr  cc_UC, init
      ret
;----- End  -----
collectData :     ; # of samples to take comes in R0
      push  R0
      mov   R0, #S_0010
      calla cc_UC, SendStr
      pop   R0
      mov   R1, #00002h
      mul   R0, R1
      calla cc_UC, waitGetChar
      mov   R8, R0
      sub   R8, #30h
      mov   R7, MDL

;- R7 contains TWICE the number of samples to take
;- and R8 contains which clock to use
collectjumpPoint:
      mov   R0, collectStart
      add   R7, R0
      mov   collectStop, R7
```

```
      mov   collectCount, collectStart
      cmp   R8, #00h
      jmpr  cc_NE, _0121
      mov   CC0IC, #00077h
      ret
_0121 :
      cmp   R8, #01h
      jmpr  cc_NE, _0122
      mov   CC4IC, #00077h
      ret
_0122 :
     cmp    R8, #02h
     jmpr   cc_NE, _0124
     mov    CC8IC, #00077h
_0124 :
     ret
;----- End collectData -----
id :
     mov    R0, #S_0011
     calla  cc_UC, SendStr
     ret
;----- End id -----

; --- string constants ----------------
even
S_0000:  db "INV", 10, 0
even
S_0001:  db "FINC", 10, 0
even
S_0002:  db "FINC", 10, 0
even
S_0003:  db "FINC", 10, 0
even
S_0004:  db "SEND", 0
even
S_0005:  db "STPA", 10, 0
even
S_0006:  db "STPC", 10, 0
even
S_0007:  db "RST", 10, 0
even
S_0008:  db "DOWN", 10, 0
even
S_0009:  db "EXEC", 10, 0
even
S_0010:  db "CLCT", 10, 0
even
S_0011:  db "BK v1.0 Loaded", 10, 0
; --- end string constants ----------------
even

collectStart:
      ret
init:
      mov   R9, #plist
      mov   R10, #00040h
      mov   R11, #000EAh
```

```
initloop:
      mov   [R10], R11
      add   R10, #2
      mov   [R10], [R9+]
      cmpi2 R10, #07Eh
      jmpr  cc_ULT, initloop
      jmpr  cc_UC, plist2
plist:
      dw    944          ; Address of isrCLK0
      dsw   3h (966)     ; Address of a reti
      dw    968          ; Address of isrCLK1
      dsw   3h (966)     ; Address of a reti
      dw    992          ; Address of isrCLK2
      dsw   7h (966)     ; Address of a reti
plist2:
      ret
; ----- include files -----------------
#include <sfr166jk.inc>
; --- end of generated code -----------
```

APPENDIX D

"USER" CODE

This Appendix contains more explanation of the Siemens 80C166 microcontroller code contained in the Code Fragments and Code Segments of the "User" code.

D.1 Setting Up for a Register Write

If a Code Segment, excluding the INIT Segment, writes to (sets) any of the Stanford Telecom STEL-1032's registers, then the following code is added to that Code Segment.

Prefixed:

Machine Language (in bytes)				**Assembly Language**	
236	249			push	R9
230	249	*daddl*	*daddh*	mov	R9, *#dataaddress*

Appended:

252	249			pop	R9

The symbols *daddl* and *daddh* stand for the low and high byte, respectively, of the beginning of the Code Segment's data (*dataaddress*). The data starts at the word address following the Code Segment's Return from Interrupt (reti).

The INIT Code Segment begins with

230	249	*daddl*	*daddh*	mov	R9, *#dataaddress*

where *daddl* and *daddh* are the same as above.

D.2 Setting the 32-bit Registers

The "Set" Code Fragment for 32-bit registers is

232	57	mov	[R3], [R9+]
232	57	mov	[R3], [R9+]
232	57	mov	[R3], [R9+]
232	57	mov	[R3], [R9+]

The corresponding data is added to the Code Fragment's data in the format

address data1 *address*+1 *data2*
address+2 *data3* *address*+3 *data4*

where *address* is the beginning address (first byte) of the STEL-1032 register being set, *data1* is the least significant byte of the 32-bit data, *address*+1 is the second byte of the STEL-1032 register being set, *data2* is the next to least significant byte, and so forth.

D.3 Setting the 8-bit Registers

The "Set" Code Fragment for 8-bit registers is

```
232    57                          mov    [R3], [R9+]
```

The corresponding data is added to the Code Fragment's data in the format

address data

where *address* is the address of the STEL-1032 register being set and *data* is the new register value.

D.4 Loading the Coders

Loading $Coder_0$ is accomplished by

```
30     226                         bclr   P3_1
31     226                         bset   P3_1
```

Loading $Coder_1$ is accomplished by

```
46     226                         bclr   P3_2
47     226                         bset   P3_2
```

Loading $Coder_2$ is accomplished by

```
62     226                         bclr   P3_3
63     226                         bset   P3_3
```

Loading $Coder_0$ and $Coder_1$ is accomplished by

```
102    226    249    255           and    P3, #11111001b
118    226    6      0             or     P3, #00000110b
```

Loading $Coder_0$ and $Coder_2$ is accomplished by

```
102   226   245   255        and   P3, #11110101b
118   226   10    0          or    P3, #00001010b
```

Loading $Coder_1$ and $Coder_2$ is accomplished by

```
102   226   243   255        and   P3, #11110011b
118   226   12    0          or    P3, #00001100b
```

Loading all three Coders is accomplished by

```
102   226   241   255        and   P3, #11110001b
118   226   14    0          or    P3, #00001110b
```

D.5 Enabling the Interrupts

The following code enables an interrupt.

```
230   x     y     0          mov   CCzIC, #y
```

where *x* is the address (188-203) of the respective interrupt's control register, *y* is the interrupt priority and sets the interrupt enable bit, and *z* is the corresponding Compare/Capture Interrupt Control Register (0-15).

D.6 Disabling the Interrupts

The following code disables an interrupt by clearing the interrupt enable bit and setting the priority to zero.

```
230   x     0     0          mov   CCzIC, #0
```

where *x* and *z* are the same as above.

D.7 Stopping Data Collection

To stop the collection of data, the corresponding Clock's interrupt is disabled with

```
230   w     0     0          mov   CCzIC, #0
```

where *w* is either 188, 192, or 196 for $Clock_0$, $Clock_1$, or $Clock_2$, respectively, and *z* is the same as above.

D.8 Starting Data Collection

Data collection is started using

```
230   247   vall   valh          mov    R7, #value
230   248   clock  0             mov    R8, #clock
202   0     30     5             calla  cc_UC, #51Eh
```

where *vall* and *valh* are the low and high byte, respectively, of twice the number of samples to be taken (*value*) and *clock* is which clock to use as the trigger (0, 1, or 2).

D.9 End of Each Code Segment

Each Code Segment for an interrupt ends with

```
251   136                        reti
```

followed by the data for any register writes.

The INIT Code Segment ends with

```
203   0                          ret
```

followed by the data for any register writes.

APPENDIX E

USER'S MANUAL

This User's Manual will help the user with the initial system setup and getting those first codes produced. It outlines the computer requirements for running the software, how to connect the hardware, how to build a Process List, and how to use the data collection capability.

E.1 Hardware/Software Requirements

A Personal Computer (PC) running Microsoft® Windows®95 with The Mathworks® MATLAB® v5.0, or later, installed is required to run the Graphical User Interface (GUI) software. At this time, the software has not been tested on a Windows®98 platform. An available RS-232 Serial Interface is required to connect the PC to the Kos/Brendle Code Generator Box (Kode Box, for short). The appropriate cable to connect the PC's RS-232 port to the Kode Box's female DB-9 connector is also required.

All of the Kode Box MATLAB® software must be located in a directory included in the MATLAB® search path. The correct COM port must be specified in the "opencomm.m" file. In the "COMMH = portcom('OPEN', 'COMx', 65000, 15000)" line, change the "COMx" to the appropriate port, either COM1, COM2, COM3, or COM4 (default setting is COM2). To determine the proper setting, consult your Windows® manual to find the appropriate available port.

For the purposes of this demonstration, a signal generator, oscilloscope, and appropriate BNC patch cables are helpful.

E.2 Hardware Connections

Connecting the Kode Box to the PC is accomplished by connecting the Kode Box's RS-232 port, found on the rear panel of the box, to the PC's RS-232 port with the appropriate cable.

Connect the signal generator output to the CLK0 input on the front of the Kode Box. Connect the CODE0 output to the oscilloscope's channel one input. Finally, connect the EPOCH0 output to the oscilloscope's channel two input.

E.3 Generating a Code

At this point, the computer should be powered up with MATLAB® running. To start developing a Process List, which will eventually control the Stanford Telecom STEL-1032 within the Kode Box, enter "plqb" at the MATLAB® prompt. The *Process List Quick Builder* (PLQB) GUI should appear.

Every register of the STEL-1032 is shown on this one GUI. Enter "23" in the edit box located to the right of "Polynomial in Octal" in the upper left corner of the GUI. By doing so, the user is specifying the generator polynomial $G = 1 + X^1 + X^3$. Next, enter "7" in the "Initial Fill in Octal" edit box and "2" in the "Epoch register" edit box.

For demonstration purposes, enter "33" in the "Phase Mux" edit box. Now, press the *Validate* button. The "Phase Mux" text will turn red, indicating an incorrect value has been entered. Upon pressing the *Validate* button, all register values are range checked and the labels of any offenders are changed to red. Change the "Phase Mux" value to "1" and press the *Validate* button again. "Phase Mux" will return to black. Be sure that the check boxes next to the "Coder1" and "Coder2" labels are *unchecked.* Place a check next to the "Mixcode" label.

All values on the PLQB GUI can be saved and recalled using appropriate options under the *Workspace* menu bar item. If you would like, test this feature by saving the values you just entered.

Pressing the *Export to PLD* button causes the Process List Developer (PLD) GUI to open and values entered in the PLQB edit boxes are automatically transferred to the PLD in the form of a Process List (PL). Press the *Export to PLD* button now.

The PLD allows the user access to all capabilities of the Kode Box microcontroller system. The user can develop, edit, store, and recall PLs capable of fully controlling the STEL-1032. By constructing Interrupt Service Routines (ISRs), the user can control the STEL-1032 in ways not possible with a stand-alone STEL-1032. The user can optionally bypass the PLQB and start with the PLD.

By "mistake," the Mixcode configuration was transferred to the PL from the PLQB. To remove that *Process List Item* (PLI), single left click on that line in the PL and press the *Delete* button to the right. Now, select the *Undo* menu item and you will see that the deleted PLI is available for reinsertion.

Now, to construct an ISR, select the following items in the Process List Item Builder (PLIB) at the bottom of the PLD screen. Under "When," select INIT. Select Enable_Int in the "Action" list and select EPOCH0 in the "What" list. Press the *Add* button. The new PLI will appear in the PL above. This line enables the EPOCH0 interrupt.

Before a Coder can begin to generate codes, the Coder must be loaded with an initial fill (starting phase). Thus, PLIs need to be added to cause this action. Select INIT in the "When" list, Load in the "Action" list, and CODER0 in the "What" list. Press the *Add* button to insert the new PLI.

Next, select EPOCH0 in the "When" list, Load in the "Action" list, and "CODER0" in the "What" list. Press the *Add* button. This PLI instructs the microcontroller to reload $Coder_0$ with the contents of the $INIT_0$ register upon an $EPOCH_0$ pulse. Although the STEL-1032 can perform this operation with built in control mechanisms, we will use the microcontroller for demonstration purposes.

The PL is now ready to be used. Press the *Done* button. *The Process List Communicator* (PLC) will now open. The PLC is used to communicate with the Kode Box. The first necessary step is to bootload the Kode Box. If the Kode Box is not already turned on, turn it on now. To accomplish the bootload, press the hardware Reset button on the front of the Kode Box and then

press the *Establish Comm* button. If the bootload is successfully accomplished, the text "Communication Established" will appear and the *Establish Comm* button will be grayed out. The *Export PL* button also becomes available.

Press the *Export PL* button to compile your PL and download it to the microcontroller. Upon a successful download, the text "PL Export Successful" will appear and the remaining pushbuttons will become available.

At this time, make sure that the "Man/Auto" switch on the front of the Kode Box is in the "Auto" position and the "Run/Stop" switch is in the "Run" position. Set the signal generator to produce a 100 kHz TTL level clock pulse.

To execute your PL and begin generating of a code, press the *Execute PL* button. With the oscilloscope set to trigger on the second channel (EPOCH0), adjust the controls as necessary to obtain a nice display of the code being generated. In this demonstration, the code should repeat every 12 clock cycles.

E.4 Output Sampling

Now that the Kode Box is producing a code, the signals can be collected, retrieved, and assigned as variables in the MATLAB® workspace. To do this, place the value "1000" (1000 samples to be collected) in the edit box above the *Start Collection* button. Ensure that "CLK0" is selected in the list box next to the edit box. Press the *Start Collection* button. The text "Data Collection Complete" will soon appear under the button.

Press the *Retrieve Data* button to upload the collected data to the PLC. Next, put check marks in the boxes next to the "CODE0" and "EPOCH0" labels. In the edit box next to the "CODE0" label, enter "code0." In the edit box next to the "EPOCH0" label, enter "epoch0." Finally, pressing the *Export Data* button assigns these signals as designated variables in the MATLAB® base workspace.

E.5 Do's and Don'ts

The following is a list of helpful "do's and don'ts."

Do:

- Do place the "Man/Auto" switch in the "Auto" position
- Do place the "Run/Stop" switch in the "Run" position
- Do load the Coders intended to produce a code by some means
- Do enable the used interrupts by some means

Don't:

- Do not enable an interrupt for which there are no associated Process List Items
- Do not use the clock selected for data collection for any other interrupt purposes

BIBLIOGRAPHY

1. Sklar, Bernard, *Digital Communications: Fundamentals and Applications*, New Jersey: Prentice Hall, 1988.

2. Brendle, John F., Jr., *Pseudorandom Code Generation for Communication and Navigation System Applications*, MS Thesis, AFIT/GE/ENG/97D-16, Air Force Institute of Technology, Wright-Patterson AFB OH, December 1997.

3. Peterson, Roger L., et. al., *Introduction to Spread Spectrum Communications*, New Jersey: Prentice Hall, 1995.

4. Dana, Peter, *Global Positioning System Overview*, University of Texas at Austin, http://www.utexas.edu/depts/grg/gcraft/notes/gps/gps.html, 1998.

5. Dixon, Robert C., *Spread Spectrum Systems*, New York: John Wiley & Sons, 1984.

6. Hersch, Russ, *Embedded Processor and Microcontroller Primer and FAQ*, http://www.cis.ohio-state.edu/text/faq/usenet/microcontroller-faq/primer/faq.html, 1997.

7. *PRN Coder STEL-1032, Data Sheet*, Stanford Telecommunications, Inc., California: 1990.

8. Tecny Electronics, http://www.tecel.com/.

9. Rigel Corporation, http://www.rigelcorp.com/.

VITA

Second Lieutenant John M. Kos was born on January 2, 1975 at Fort Bragg, NC. He graduated from Bedford North Lawrence High School in Bedford, IN, in 1993 and attended the Rose-Hulman Institute of Technology in Terre Haute, IN, on an Air Force Reserved Officer Training Corps (AFROTC) scholarship for electrical engineering. As a member of AFROTC Detachment 218, he earned the rank of Cadet Major and actively participated in the honorary Arnold Air Society (AAS), earning the rank of AAS Cadet Major as the Squadron Commander. Upon graduation, he was awarded a Bachelor of Science degree in electrical engineering and was commissioned into the USAF Active Reserve.

His first assignment on active duty was to the Air Force Institute of Technology, Wright-Patterson Air Force Base, OH. As a Masters candidate in the Department of Electrical and Computer Engineering, his focus of study was on communications and computer networks.

After graduation, he will be assigned to the Air Force National Air Intelligence Center at Wright-Patterson AFB, OH.

Permanent Address: 7 Taylor Run, Bedford, IN 47421

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 1999	Master's Thesis

4. TITLE AND SUBTITLE
Graphical User Interface and Microprocessor Control Enhancement of a Pseudorandom Code Generator

5. FUNDING NUMBERS

6. AUTHOR(S)
John M. Kos
2nd Lieutenant, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology
Wright-Patterson AFB, OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GE/ENG/99M-15

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
James P. Stephens, Sr.
AFRL/SNRW Bldg 620
2241 Avionics Circle Room N3-F10
Wright-Patterson AFB, OH 45433-7333

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*
Modern digital communication techniques often require the generation of pseudorandom numbers or sequences. The ability to quickly and easily produce various codes such as maximal length codes, Gold codes, Jet Propulsion Laboratory ranging codes, syncopated codes, and non-linear codes in a laboratory environment is essential. This thesis addresses the issue of providing automated computer control to previously built, manually controlled hardware incorporating the Stanford Telecom STEL-1032 Pseudo-Random Number (PRN) Coder. By incorporating a microcontroller into existing hardware, the STEL-1032 can now be conveniently controlled from a MATLAB® Graphical User Interface (GUI). The user can quickly create, save, and recall various setups for the STEL-1032 in an easy to use GUI environment. In addition to having complete control of the STEL-1032's internal actions, the microcontroller adds an extra measure of control possibilities by using various signals as possible interrupt sources. The microcontroller can sample the STEL-1032's various outputs at a rate up to 320 kHz and the data can be imported directly into MATLAB® for further analysis.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Pseudorandom Code, Spread Spectrum, Code Generator, Microcontroller, Graphical User Interface	113
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL