B. Tech Project Report

Microcontroller Based Power Distribution Monitoring & Control

Project Guide: Dr S. P. Das

Submitted By -

Alekh Jindal (Y2038)
Yash Agarwal (Y2436)

Acknowledgements

We would like to express our sincere gratitude to Dr S. P. Das, Associate Professor, Department of Electrical Engineering, IIT Kanpur, for his continuous encouragement and support. During the topic finalizing period he helped us fix a target and then it was real fun chasing it. Dr Das initiated us with a very simple approach and thereon building the complexity as the project moved on. We are grateful to him for giving us such a wonderful learning experience.

We are also highly grateful to Mr. Kapil Deo, In-charge, Microprocessor Lab, for his support and cooperation over strenuous long hours. It was due to his urge and motivation that we were able to realize this project.

We are also thankful to Mr. J Phani Kumar, M. Tech Student, and to Mr. Ranjan Kumar Behara, PhD Student, for their useful help and support.

<u>Contents</u>

1. Introduction
1.1 Motivation
1.2 Project Idea
1.3 Reinventing the wheel
1.4 Project Implementation
1.5 Organization of the Report

2. Overall Project Design

3. Remote Terminal Unit
3.1 Real Time Sampling
3.2 Computations
3.3 Digital output
3.4 Communication Handler

4. Base Station

5. Distributed Network Protocol(DNP) 3.0
5.1 Introduction
5.2 Design
5.3 Implementation
5.4 Layering

6. Schematics of Wireless Communication

7. Cyclic Redundancy Check(CRC)
7.1 Objective
7.2 Algorithm
7.3 Hardware Implementation
7.4 Software Implementation

8. Experimental Results and Discussions
8.1 Performance
8.2 Source of Error
8.3 Snapshots of work

9. Conclusion

10. References

Appendix A Assembly Codes
Appendix B LabVIEW Codes

Abstract

This project aims to automate the process of obtaining data relating to power distribution such as from a substation or transformer and to control a circuit breaker, a motor, or a valve. The system consists of a Base Station and a Remote Terminal Unit (RTU). Base Station is a Desktop PC running a graphical user interface (GUI) generated in labVIEW. RTU is 80196KC microcontroller-based Terminal which is located at a remote substation / transformer. The RTU being compact can even be installed on the pole top of a transformer. It measures the voltages and currents of input lines, and calculates RMS values of voltages and currents along with average power of the line. The Base Station and the RTU communicate via serial link through RS-232 ports. The physical medium can be wired or wireless using BiM-418-F transceiver chips.

The GUI at Base Station allows user to monitor RMS voltage, current and average power of any line. Monitoring can be instantaneous or in continuous mode. Thereafter energy audit and other forms of analysis can be carried out at the base station, where data from all the different substations will be available. In addition to this, there is also some control capability at the base station. The operator can operate on digital lines i.e. switch on or switch off any breaker connected to the RTU.

To ensure validity of exchanged data and standardization in the process, distributed network protocol (DNP 3.0). It helps in overcoming noise and signal distortion. DNP3 software is layered to provide reliable data transmission. Layering also helps to organize the transmission of data and commands. The three layers in DNP3 protocol are: *application layer*, *data link layer* and *physical layer*. The project implements Multi-drop DNP3 architecture whereby one master communication device (Base Station) is connected to several slave communication devices (RTUs). To ensure reliability, error detection algorithm cyclic redundancy check (CRC) is used. It is used along with DNP 3.0 protocol. CRC algorithms are designed to maximize the probability of error detection. The probability that a message contains errors and the CRC stills checks out is very low.

Key words: Intel 80196 microcontroller, LabVIEW 7.1, Power Distribution System, Remote Terminal Unit, DNP 3.0 protocol, Base Station

1. Introduction

1.1 Motivation

The motivation for this project comes from the need for an efficient system of energy management. Efficient power distribution requires interactive monitoring and control of the distribution/transmission network. Moreover in India, a substantial portion of energy is drained by unauthorized power consumption, thereby requiring further attention. In order to cope up with increasing demand of reliable and quality power, there is a need for automated maintenance with provisions for dealing with cases of failure.

1.2 Project Idea

This project aims to provide an automated system whereby energy flow can be closely monitored and controlled remotely. The plan is to come up with an integrated microcontroller based wireless remote terminal unit. The terminal unit would be operating in the actual field setting and would be concerned with monitoring and control of the distribution network. This necessitates the terminal unit to be integrated and robust. The terminal unit would be operated from the base station via user friendly software tools. This provides the facility of post processing and analysis centrally and in a more rigorous manner. This will enable the detection of distribution bottle necks and will also account for the high losses that are being incurred.

A two way wireless communication link would be used to communicate between the terminal unit and the base station. Base station links together several terminal units and hence acts as a central server for the different power distribution links. The central nature of the base station is specifically useful since network wide view of the power distribution can be visualized. This can help in taking actions on a part of network due to events occurring on some other part of the network.

1.3 Reinventing the Wheel

A similar solution has been previously proposed but it differs from this project in the following sense –

1. This project focuses on an integrated remote terminal unit (RTU). A 16 bit microcontroller instead of a PC based setup can suffice the purpose of RTU controller.
2. The RTU is expected to be typically $1/4^{th}$ the size of the existing solution.
3. Given its robustness, this RTU can be installed at pole tops or at locations remote in real sense.
4. Wireless communication link is being established which uses distributed network protocol (DNP 3.0).

5. A centralized base station (for all RTUs) with user friendly graphical user interface (GUI) is being provided to the end user.
6. It is a low cost solution.

1.4 Project Implementation

Remote Terminal Units (RTUs) will be mounted on all transformers distribution network. These RTUs would have the capability to measure line currents and line voltages (through the use of Current Transformers and Potential transformers). They would then transmit this data, on a periodic basis, to a central base station, which would be located in the substation. Each distribution transformer would have complete accountability for the power that it is extracting from the grid. Similarly, every substation would have complete information about power flow in its part of the grid.

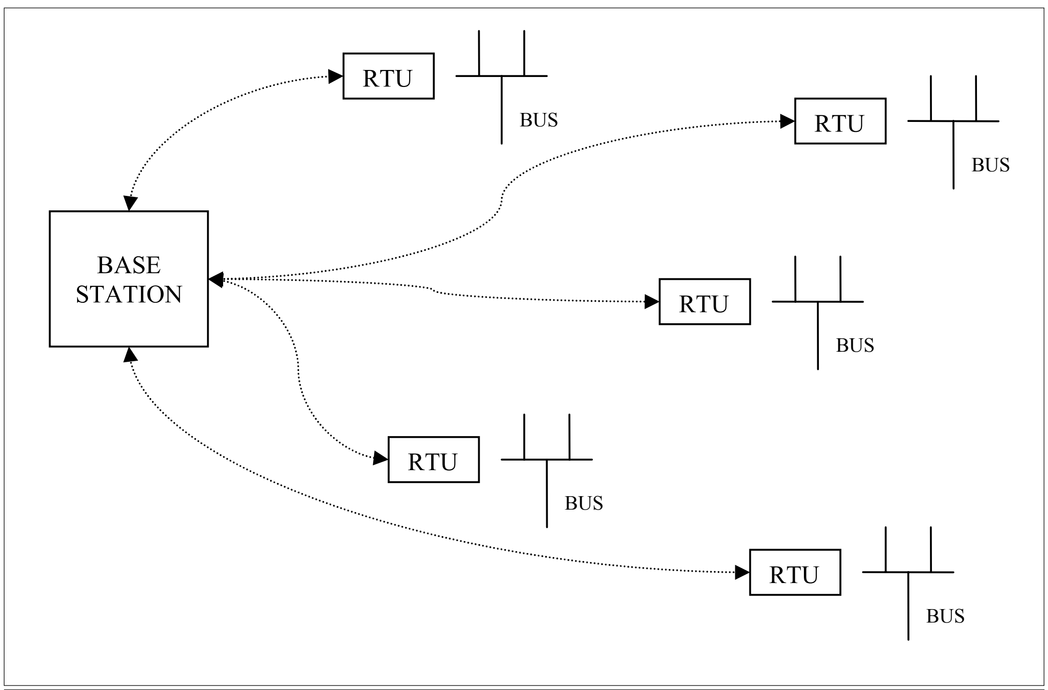


Fig 1.1 - Project Schematic

1.5 Organization of the Report

Following this brief introduction about project idea, motivation and implementation we have the subsequent report organized as follows.

Overall Project Design: It elaborates the project design and the main components of the system.

Remote Terminal Unit : This section discusses how the various functions (like real time sampling, A/D conversion and so on) at the RTU have been implemented using the Intel 80196 microcontroller.

Base Station: The Graphical User Interface (GUI) that has been developed using LabVIEW is explained here.

Distributed Network Protocol (DNP) 3.0: This protocol is being implemented in this semester and here it is explained in detail. Along with the design, the implementation of DNP in the project is discussed. Lastly the three layers of the protocol are explained along with discussion on cross-layer communication.

Wireless Communication: This proposes a schematic for wireless communication in place of wired one. The worked out details and issues of setting up wireless communication are put forth.

Cyclic Redundancy Check (CRC): Here an error detecting algorithm which has been used to ensure reliable communication is discussed. Also, hardware and software implementation are of this algorithm are shown.

Experimental Results and Discussions: The hardware setup along with performance analysis is shown. The measurement and computation results are also compared with the calculated ones. Also, snap shots of the work with discussion is presented.

Conclusion: Presents summary of the work and suggests some scope for future work.

Appendix A contains the assembly codes written for 80196KC microcontroller.
Appendix B contains the labVIEW codes written for the Base Station.

2. Overall Project Design

The system has three major components –

1. ***Remote Terminal Unit:*** It is equipped with the tasks of real time sampling of current and voltage signals, computation of root mean square values of current and voltage along with average power, digital output for line or breaker control and handler for communication with the base station.

2. ***Communication Setup:*** Communication is done over wireless medium using distributed network protocol (DNP). Both the RTU and the base station use a transceiver module for this purpose. A two way communication link provides the interactivity between the two end points.

3. ***Base Station:*** This is aimed at monitoring and controlling the RTU remotely. Apart from the communication handler, base station has a user friendly graphical interface. It allows user to monitor instantaneous as well prolonged responses from one or more RTUs.

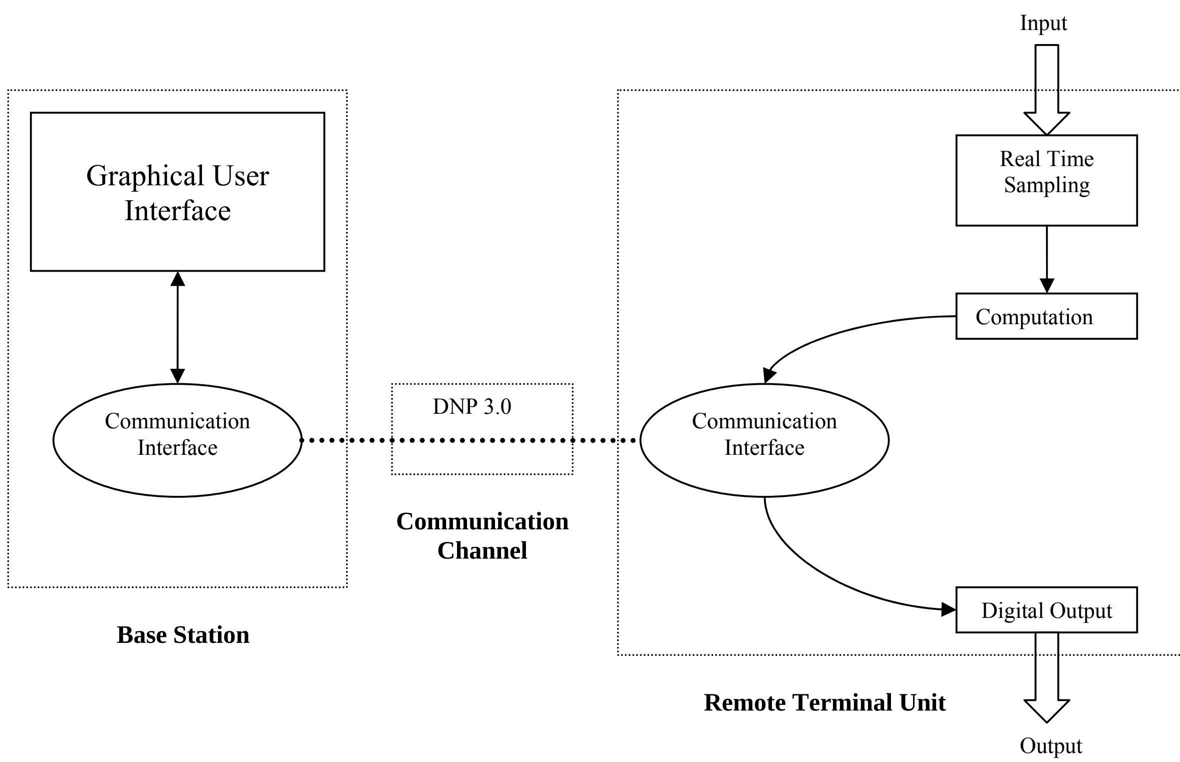


Fig 2.1 - Block Diagram of the design

3. Remote Terminal Unit

3.1 Real Time Sampling:

High Speed Output unit (HSO)

The HSO triggers events at specific times with minimal CPU overhead. Events are generated by writing commands to the HSO_COMMAND register and the relative time at which the events are to occur into the HSO_TIME register.

```
LDB HSO_COMMAND, #what _to_do
LDB HSO_TIME, #when_to_do_it
```

Events can be based on Timer1 or Timer2, such that whenever HSO_TIME matches the timer value the event loaded into HSO_COMMAND is triggered. Up to 8 events can be loaded into the HSO control at one point of time. More ever, normally the events are cleared from the HSO control once the events are triggered. We have to lock them into the HSO control such that they occur repeatedly until stopped.

Timing Considerations

This sampler samples 16 samples per sample for 4 cycles and then computes the rms and average values.

Assuming input frequency to be 50 Hz,
Time between two successive samples = (1/50) * (1/16) sec
= 1.25 ms

Internal operation is based on the oscillator frequency divided by two, giving the basic time unit known as 'state time'. Given 12 MHz crystal on the kit –

State time = (2/12) us = 166.66 ns

Now, since up to 8 events can be loaded into the HSO control, it takes the HSO control 8 state times to compare all HSO_TIMEs with the timer value. In order to avoid missing any of the events, it is desirable to make the timer increment every 8 state times.

Therefore, time between two successive timer counts = 8*166.66ns
= 1.33 us

Hence, number of timer counts required between two successive samples = 1.25 ms/1.33us
= 937.5
~ **938**

<u>*Note*</u>: Analog to Digital converter taker 158 state times for full conversion i.e. (158/8 ~ 20) timer counts occur while the digital conversion is in process. Hence the converter is ready by the time the timer reaches 938 the next time.

Thus, based on this we need to feed the following two events into the HSO control –

a) Start A/D sampling after 938 timer counts
b) Reset timer on 939th count

This sampler uses Timer2 for HSO timing. IOC2 sets Timer2 to count every 8 state times and to count up. IOC0 resets Timer2. T2CNTC sets to clock Timer2 internally.

A/D converter

AD_COMMAND is used feed the Analog to Digital converter. It is set to give 8 bit digital output. The converter may be instructed to start sampling immediately or when it is triggered by the HSO. This sampler uses A/D converter in the latter mode. However if more than one A/D conversions are to be made then the converter needs feeding repeatedly through AD_COMMAND. A/D conversion complete interrupt is enabled using INT_MASK. The address of the Interrupt Service Sub-routine (ISR) is stored at 6002h which is where the processor looks for the A/D complete ISR address. A/D result is read from AD_RESULT register. Currently it samples 64 samples each from 8 input channels in a cyclic fashion.

Summary of registers used by the real time sampler is given below –

HSO_COMMAND	Load events into High Speed Output unit
HSO_TIME	Load the triggering time corresponding to the respective event
IOC0.0	Reset Timer2
IOC2.0	Timer2 counts every 8 state times
IOC2.1	Count up
IOC2.6	Enable locking of HSO commands
T2CNTC	Clock Timer2 internally
AD_COMMAND	Activate A/D converter
AD_RESULT	A/D result
INT_MASK	To enable the A/D conversion complete interrupt

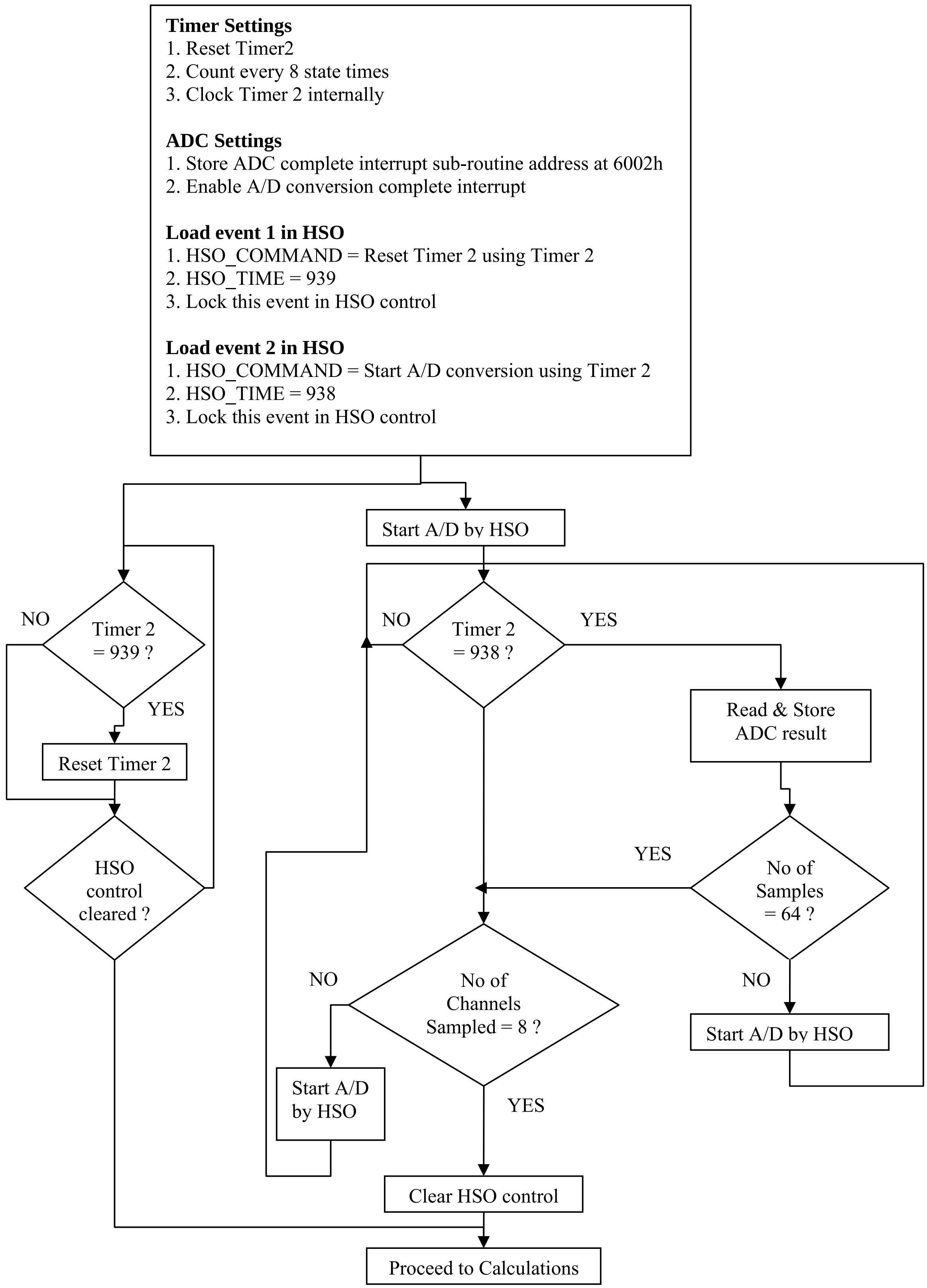


Fig 3.1: Real Time Sampling Flow Chart

3.2 Computations:

Root Mean Square

Each of the analog channel needs to computed for its root mean square (*rms*) values because –

a) It is easier to deal with *rms* values rather than with individual samples.
b) Since the processing speed of the microcontroller is much faster when compared to the transfer rates of the communication interface, this increases the overall efficiency.
c) Transmitting the large number of samples to the base station creates lot of traffic in the communication medium.
d) Burdening the base station with processing overhead is not a nice idea.

The equations used to calculate the above are given below –

$$V_{rms} = \sqrt{\frac{1}{n}\sum_{i=1}^{n-1} V_i^2}$$

$$I_{rms} = \sqrt{\frac{1}{n}\sum_{i=1}^{n-1} I_i^2}$$

The multiplication, addition and squaring operations can be easily carried, as these are part of the instruction set of 80196. The divide operation is also trivial as the division is only by 64. It is achieved by shifting the bits to the right six times. The operation of interest is the square root function as this is not a standard function in the 80196 instruction set.

A look up table containing squares of all integers from 0 to 255 has been stored in memory. First the number is compared with all the squares to determine the interval between which it lies. A linear approximation is made to calculate the square root of the number. Let x be the number whose square root needs to be calculated and it falls between x1 and x2 (y1 and y2 are their respective square root). Then the square root of x by this method will be

$$y = y1 + \frac{x - x1}{x2 - x1}$$

The error involved is within a few percent for the numbers concerned. Since we are just calculating up to 2decimal places, the use of this method is justified. Let us take an example of 110.

Actual Square root = 10.49
Square root by this method = 10.48

The following graph shows the differences between the actual and approximated values of square roots. Note that the approximation is more erroneous for the lower end of the number line, where the difference is the most.

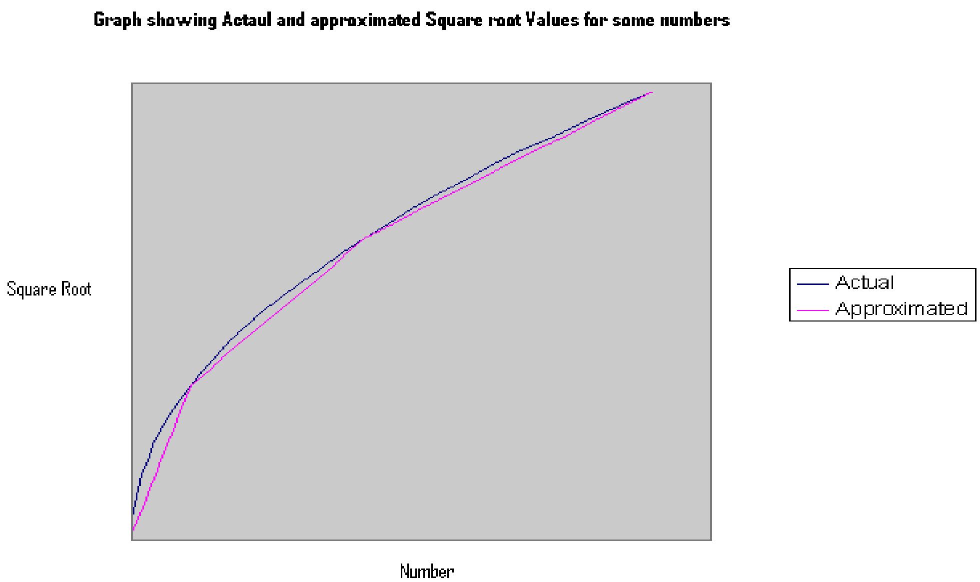


Fig 3.2: Comparison actual and approximated square root function

Average Power

The real time sampler assumes input signals to be alternately voltage and current signals as follows –

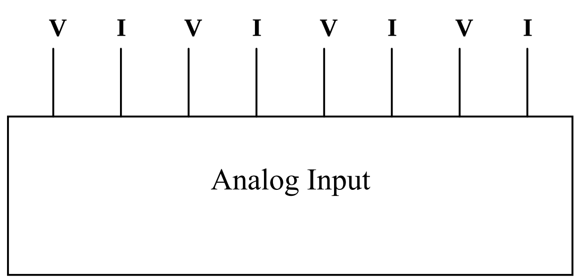


Fig 3.3: Analog Input Arrangement

With this assumption we calculate the average power and power factor as follows –

$$P_{avg} = \sum_{i=1}^{n-1} V_i I_i$$

$$PF = \frac{P_{avg}}{V_{rms} I_{rms}}$$

3.3 Digital Output:

In addition to sampling and carrying out calculations, there is also a feature for remote control. The idea here is to completely eliminate the need of an operator at the transformer or breaker. All the readings / operations should be possible from remote locations.

There are currently eight output channels (single bit). These can be used to operate eight feeders. A high on the bit line would switch on the breaker and a low on the bit line would switch off the breaker.

3.4 Communication Handler:

The RTU communicates with the base station via serial communication port. Currently the communication is wired. Communication is two way and involves the following exchanges of messages –

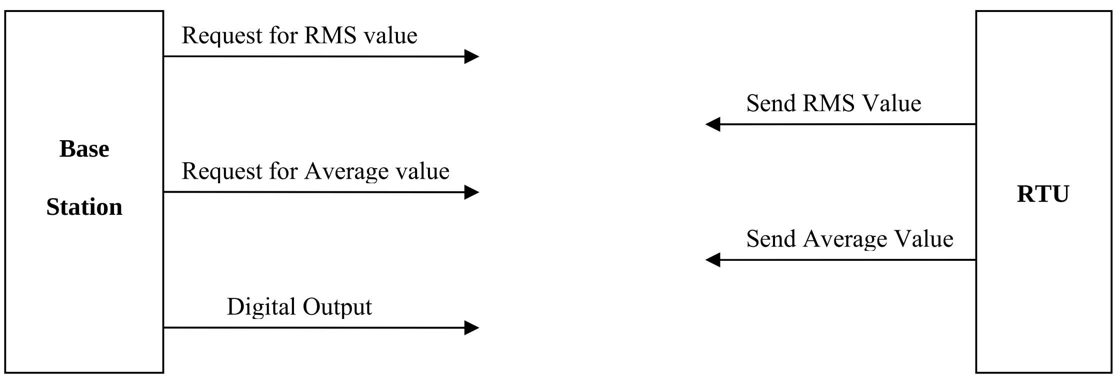


Fig 3.4: Interaction between Base Station and RTU

The frame format used for the requests is as follows

Bit 0 – Not used
Bit 1 - Defines whether it is a write or read operation. 1 – Write, 0 – Read
Bit 2 – For read it defines whether the read is Pavg or Vrms / Irms
For write it is not used
Bit 3 - Bit 7 – specify the channel

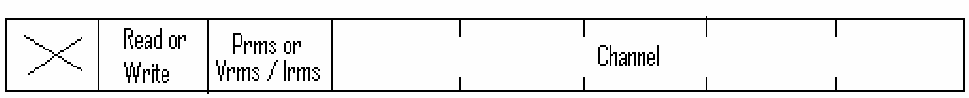


Fig 3.5: Frame Format of exchange message

The microcontroller kit SBC – 196 comes with an in-built serial port. The procedure is as follows:

- An incoming request from the base station through the serial port and triggers the serial port interrupt.
- In the interrupt service routine, the incoming is request is analyzed and the required data is transmitted through the serial port
- After the interrupt has been serviced, the microcontroller returns to the start of the sampling routine and not to its position before the interrupt. This is essential to maintain proper timing for the sampling routines and to avoid any discontinuity in the samples.

Settings

Serial port is set to operate in asynchronous mode 3. Baud rate is set to 2400. The registers affected are –

BAUD_RATE	Selects serial port baud rate and clock source
SP_CON	This register selects the communication mode and enables or disables the receiver, even parity checking.
IOC1.5	Enables the TXD function of P2.0
INT_MASK	Enables serial port interrupt

Value fed to BAUD_RATE is computed as –

$$\text{BAUD\_VALUE} = \frac{\text{Fosc}}{\text{Baud Rate} \times 16}$$

For baud rate of 2400, BAUD_VALUE comes out to be 312.5 (139h). To select internal clock bit 15 of BAUD_RATE is 1. Hence the value fed to BAUD_RATE is 8139h.

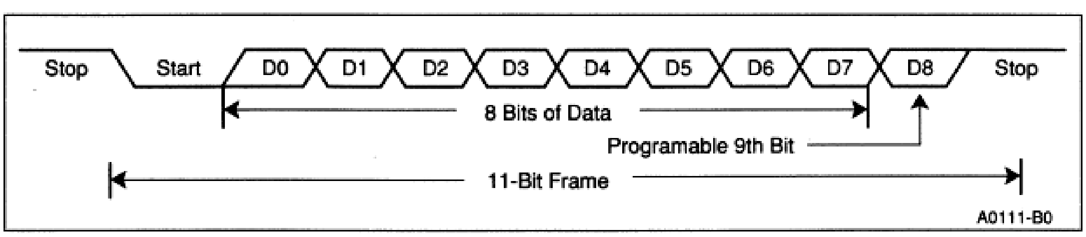


Fig 3.6: Frame Format for communication

For transmission of data from the microcontroller to the base station the format is simple. Pavg, Vrms and Irms are all 2 byte responses. For RMS values the first byte represents the number and the second byte represents the decimal. They are transmitted as “byte.byte”. For average values the entire word represents a value.

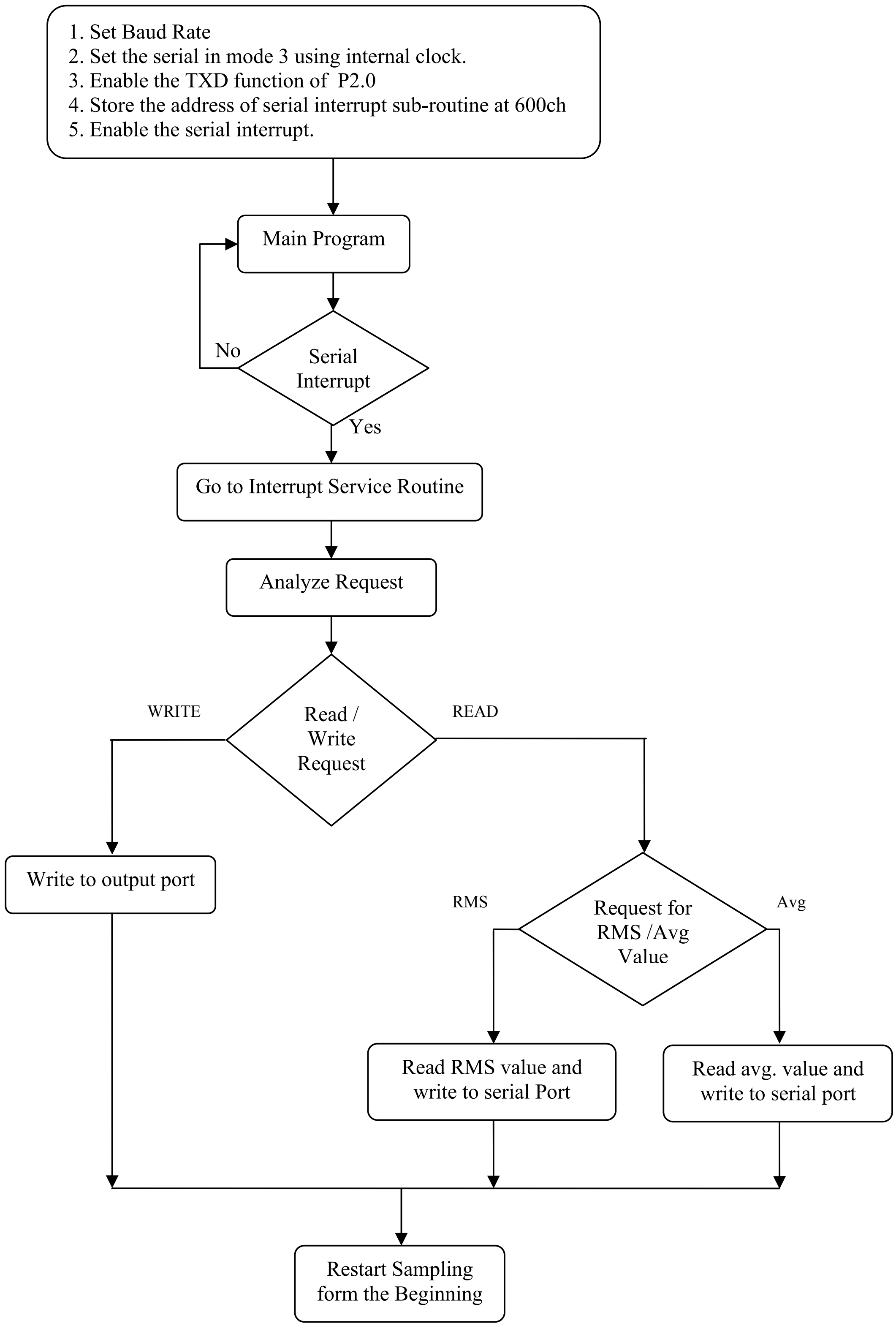


Fig 3.7: Flow Chart of serial communication handler at the RTU

4. Base Station

Labview, a software package from National Instruments, has been used for programming the base station. Essentially, Virtual Instrumentation Software Architecture (VISA) library is being used to communicate with the serial port. Currently a wire runs from Computer serial port to RTU serial port.

Base station sends requests for the following three operations –

1. To read RMS values of voltage and current.
2. To read average value of power.
3. To write the digital output.

The frame format used for the requests is as follows -

Bit 0 – Not used
Bit 1 - Defines whether it is a write or read operation. 1 – Write, 0 – Read
Bit 2 – For read it defines whether the read is Pavg or Vrms / Irms
For write it is not used
Bit 3 - Bit 7 – specify the channel

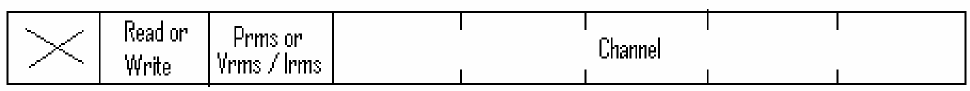


Fig 4.1: Frame Format of exchange message

Currently the frame format supports 32 Inputs / Outputs. Based on the three operations the graphical user interface provides the following three modes –

1. Update the values of Vrms, Irms, Pavg and power factor of a line.
2. Monitor Vrms, Irms, Pavg and power factor of a line in a continuous fashion.
3. Send digital output to a line.

A line here refers to two inputs (voltage and current) and one output.

After sending request for reading RMS or average values, the GUI waits for the RTU to respond. RTU responds with two bytes which are interpreted by the GUI as follows

a) If the reply is for RMS request, then the first byte is interpreted as the decimal part and second byte as integral part i.e. "byte1.byte2"
b) If the reply is for average value request, then the first byte is interpreted as the lower byte of the number and the second byte as the upper byte i.e. "byte byte"

The byte value received at base station serial port is converted into 2 hexadecimal characters. The ASCII character represented by this byte is read by Labview. Using the various subroutines in Labview this is then converted into a meaningful decimal value. In continuous mode the program continually sends requests and obtains fresh data every 3 seconds.

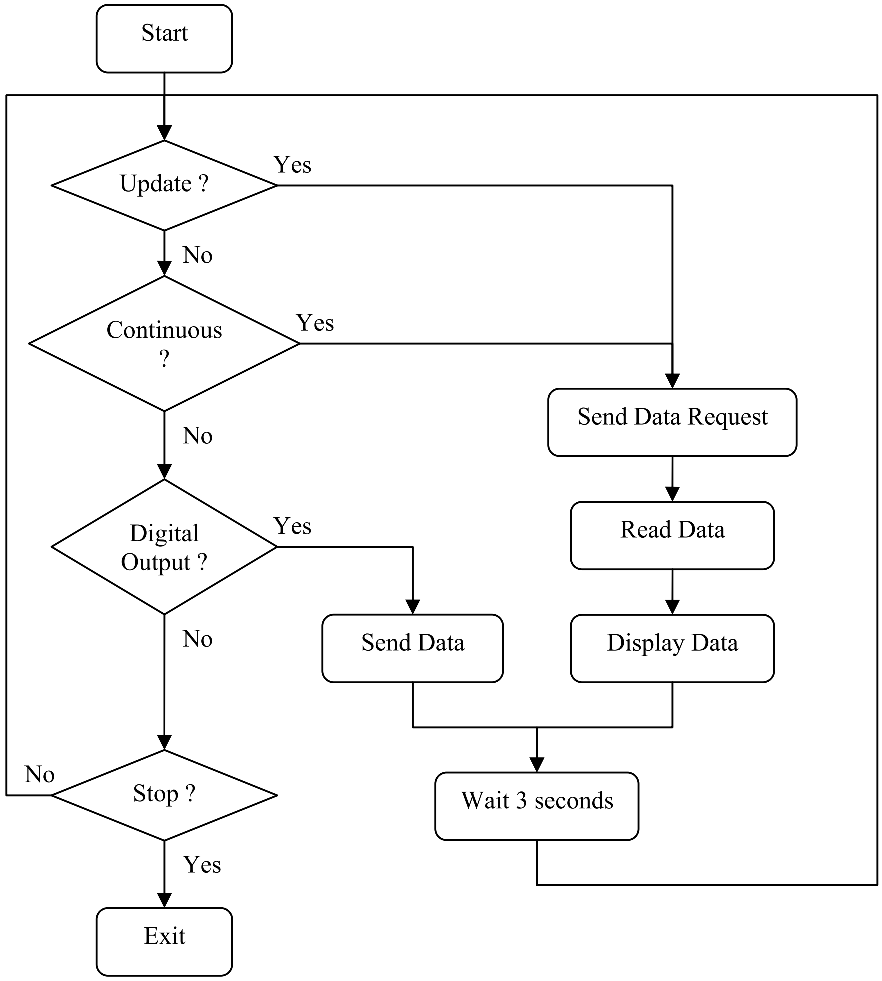


Fig 4.2: Graphical User Interface (GUI) flow chart

5. Distributed Network Protocol (DNP) 3.0

5.1 Introduction

Protocols define the rules by which devices talk with each other. DNP 3.0 is a protocol for transmission of data from point A to point B using serial and IP communications. It provides rules for *substation computer* and *remote terminal unit* (RTU) to communicate data and control commands. Data communication may involve transfer of analog input data that conveys voltages, current and power. Control commands may be to close or trip a circuit breaker, start or stop a motor, and open or close a valve. DNP3 is intended for Supervisory Control and Data Acquisition (SCADA) applications. Some of the features of DNP are –

1. Secure configuration/file transfers
2. Addressing for over 65,000 devices on a single link
3. Time synchronization
4. Broadcast messages
5. Data link and application layer confirmation

5.2 Design

Communication circuits between the devices are often imperfect. They are susceptible to noise and signal distortion. DNP3 software is layered to provide reliable data transmission. Layering also helps to organize the transmission of data and commands. DNP3 was originally designed based on three layer of the OSI seven-layer model: *application layer*, *data link layer* and *physical layer*. The physical layer defines most commonly a simple RS-232 or RS-485 interface.

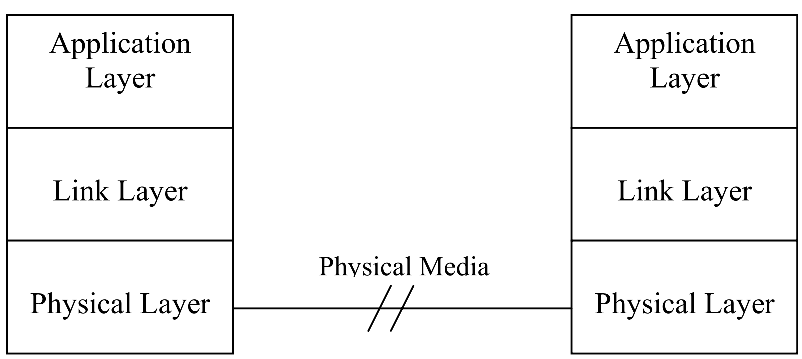


Fig 5.1 - Three layer model of DNP 3.0

Few typical system architectures where DNP3 is used are –

1. One-on-One
2. Multi-drop
3. Hierarchical

5.3 Implementation

Multi-drop system architecture is being used for the purpose of this project. Here one master station (called substation) communicates with multiple outstation devices (called RTU). It was implemented as follows –

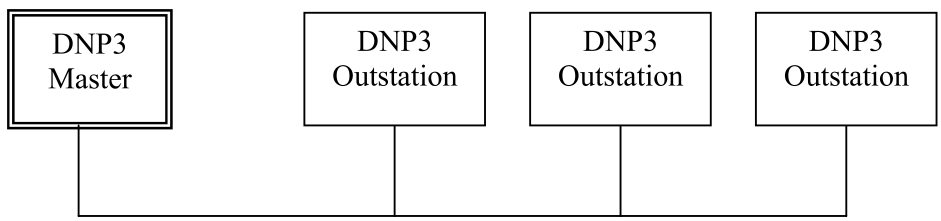


Fig 5.2 – Multi-drop DNP3 system architecture

5.4 Layering

Data communication was layered into the DNP protocol as follows –

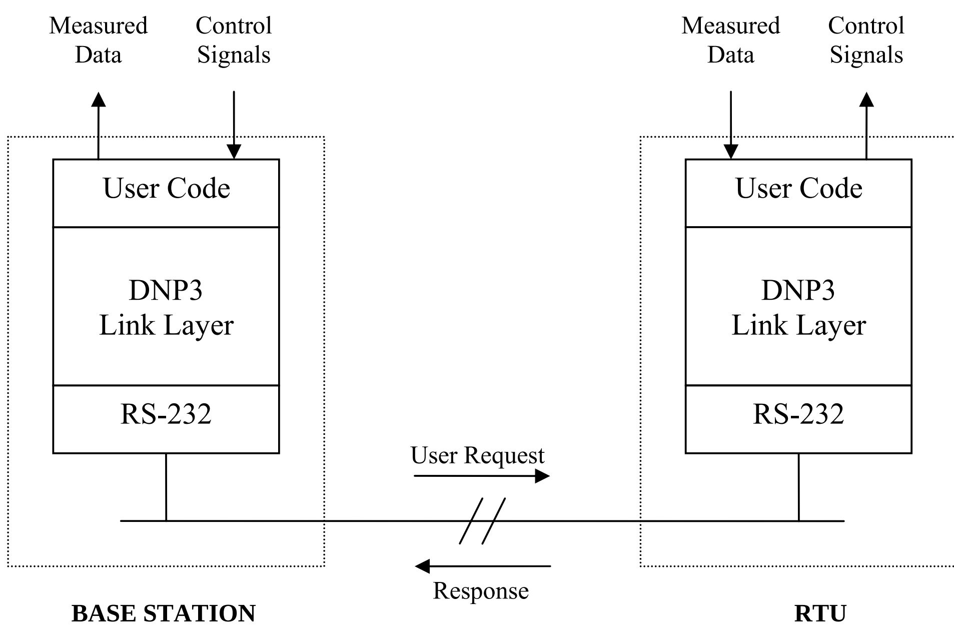


Fig 5.3 – Communication layers and data flow

Data exchanged:

1) Measured Data - is gathered by remote terminal unit (RTU) and sent over to the base station. These include RMS voltage, RMS current and average power.

2) Control Signals - are issued by base station to RTU. These include digital logic bits at the RTU end.

5.4.1 User Code Layer

User code layer processes the measured data and the control signals. User code at base station creates user interface and creates request to be sent over to the RTU. User code at RTU processes the user request and responds to it.

5.4.2 DNP3 Link Layer

Link layer receives data from User Code Layer and has the responsibility of making the physical link reliable. It does this by providing framing of data, error detection and duplicate frame detection. Link layer sends and receives packets which are called frames. DNP3 frame consists of a header and data section as follows –

DNP3 Frame

Header	Data

The header specifies the frame size, contains data link control information and identifies the DNP3 source and destination device addresses. The data section is commonly called the ‘payload’ and contains data (measured data or control signals) passed down from the layers above i.e. the User code layer. The header was implemented as follows –

Header

Sync	Length	Link Control	Destination Address	Source Address	CRC

- *Sync* – It constitutes two ‘synchronize’ bytes that help the receiver identify where the frame begins. The byte we have used is 01111110.
- *Length* – It is one byte parameter which specifies the number of bytes of data excluding CRC bytes, attached along with the frame.
- *Link Control* – It consists of a single byte which is used by the sending and receiving link layers to coordinate their activities. In this implementation receiver while responding back sends an acknowledgement byte (10101010) in the link control parameter.
- *Destination Address* – It specifies the DNP3 device for which this frame of data is intended. Only this particular DNP3 device is supposed to process

the data. It is a two byte address which implies there could be 65536 possible addresses. 12 addresses are reserved and hence 65520 individual addresses are available.

- *Source Address* – It specifies which DNP3 device sent the message. It is again two bytes long. This enables the receiver to know where to direct its response.
- *CRC* – It stands for Cyclic Redundancy Check and is used for detecting communication errors in the header. This implementation uses one byte CRC for DNP3 frame header.

The data part of the DNP3 frame contains CRC checks for every 16 bytes of data. However the last remaining chunk of data which may be less than 16 bytes also has CRC. Maximum data payload in one frame is 250 bytes excluding the CRC checks. Following is how a data frame is organized.

Data

Data (16 bytes)	CRC	Data (16 bytes)	CRC		Data	CRC

In this implementation the base station has been assigned the address FF H and RTU has been assigned the address 01 H. With this a request sent from base station to the RTU looks like –

Sync	Length	Link Control	Destination Address	Source Address	CRC	Data	CRC
2 Byte	1 Byte	1 Byte	2 Bytes	2 Bytes	1 Byte	1 Byte	1 Byte
01111110 01111110	00000001	11111111	00000000 00000001	11111111 11111111	00111001	Data/Control	CRC

If the request sent by the base station is 'data request' then the RTU responds back as follows -

Sync	Length	Link Control	Destination Address	Source Address	CRC	Data	CRC
2 Byte	1 Byte	1 Byte	2 Bytes	2 Bytes	1 Byte	2 Bytes	1 Byte
01111110 01111110	00000010	10101010	11111111 11111111	00000000 00000001	11100111	Data value	CRC

5.4.3 Physical Layer (RS-232)

It receives the frame from the Link Layer and the encoding and modulation of data. RS-232 serial communication is used as the physical layer.

Encoding: RS-232 uses Non-Return to zero (NRZ) encoding. In NRZ encoding logical '0' is represented by one line state and logical '1' by another. Data transmission starts with a START bit which is logical '0' and ends with a STOP bit which is logical '1'.

START	DATA	STOP

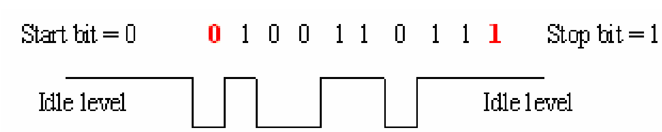


Fig 5.4 – Data transmission in RS-232

RS-232 inverts the signals and so logical '0' is +10V while logical '1' is -10V. The driver and receiver logic level is shown below.

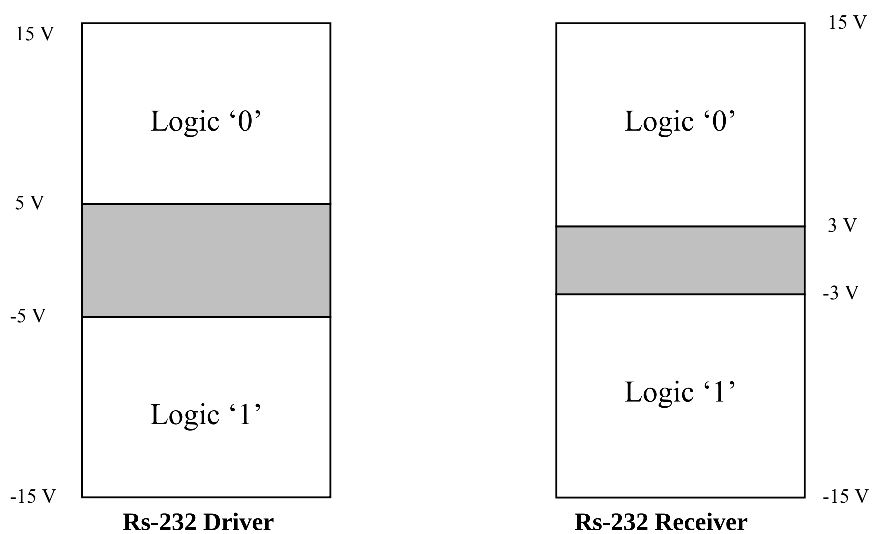


Fig 5.5 – RS-232 driver and receiver logic level

Physical Medium: The implementation uses wired medium for communication. For wireless communication BiM-418-F transceiver chips can be used. It provides low cost solution to implement a bi-directional short range radio data links.

6. Schematics of Wireless Communication

Wireless communication capability is provided by BiM-418-F transceiver chips. Salient features of this transceiver chip are –

- 30 meter range without buildings
- Single 4.5-5.5 supply
- Half duplex at up to 40 KBits/s
- 418 MHz

The transceiver being half duplex, it can transmit and receive one at a time. Thus we need an additional bit to set the communication mode (receive/transmit) of the transceiver both at the base station and the RTU. Two pins TX Select and RX Select are provided in the transceiver for this purpose. They can be configured as followed –

TX Select	RX Select	Operation
1	1	Power Down
1	0	Receiver Enabled
0	1	Transmitter Enabled
0	0	Self-test Loop

The communication setup between the base station and RTU is shown below -

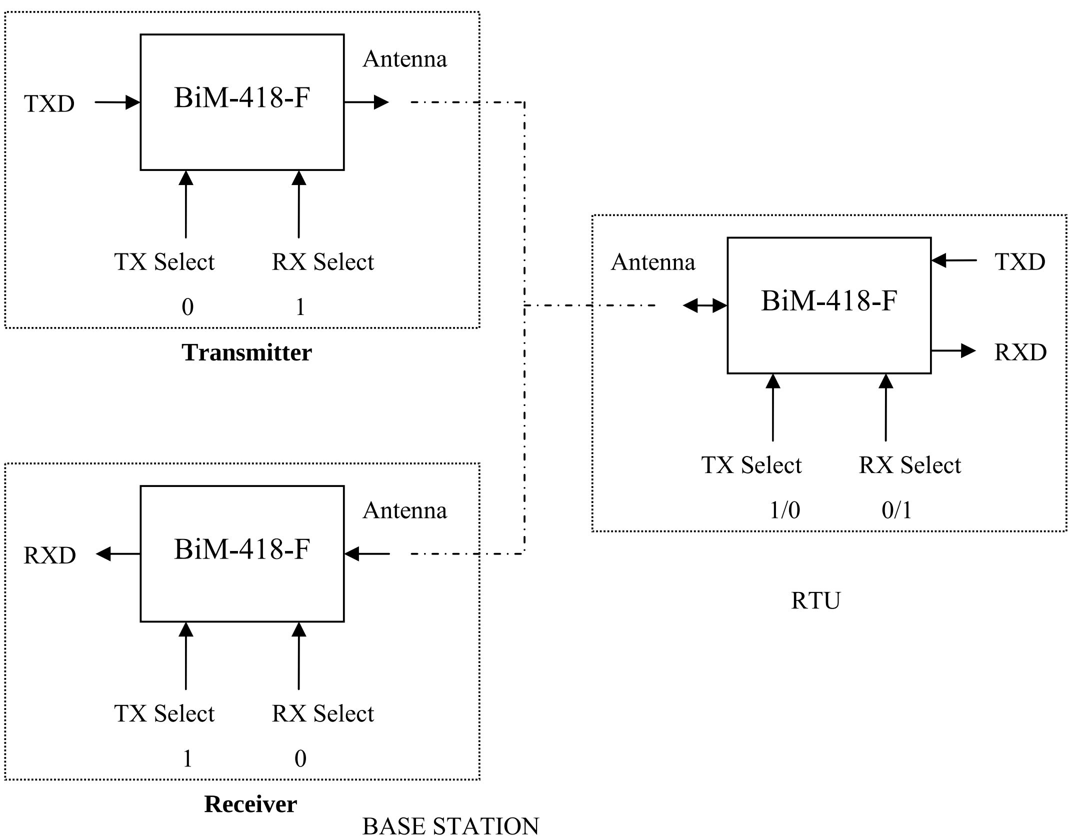


Fig 6.1 – Wireless communication setup

The RTU has one transceiver for reception followed by transmission of data. Whereas the base station has one transceiver dedicated for transmission and another dedicated for reception. This makes communication at base station to be *full duplex*. Thus, the communication setup allows the base station to issue requests to some RTU while receiving data from some other RTU at the same time. The RTU however doesn't need to be full duplex as it just responds to the user requests from the base station.

The TXD pin of the transceiver operates in the range 0-5V whereas the RS-232 driver encodes data into +10V / -10V signals. Thus +10V/-10V signal from the RS-232 port needs to be converted to 0V/+5V (since logical '0' is +10V in RS-232). The following circuitry was designed for this purpose.

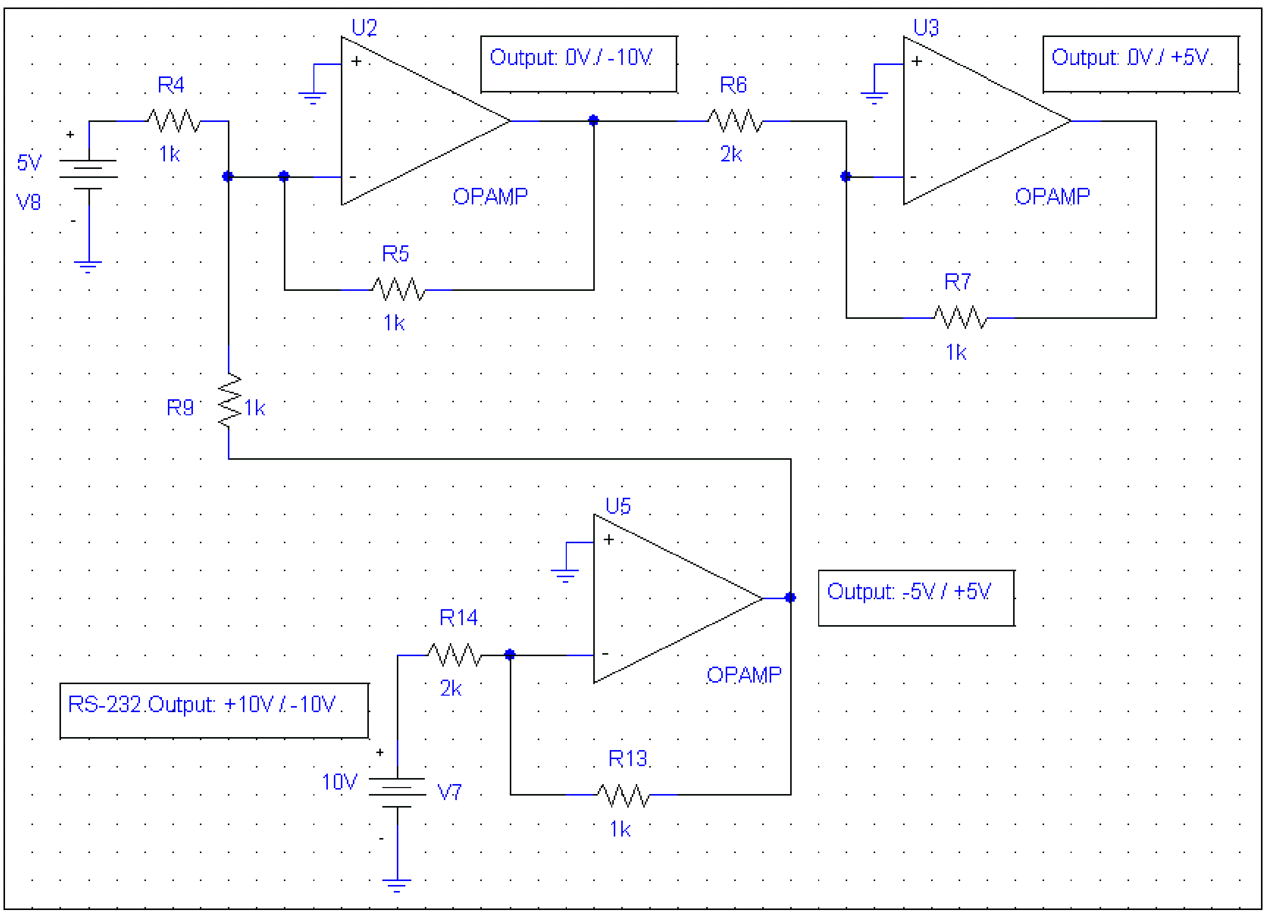


Fig 6.2 - Circuit Schematic to feed RS-232 output to Transceiver TXD pins

Similarly the RXD pin of the transceiver produces digital output 0/5 V. Before feeding this signal to RS-232, we need to convert it to +10V/-10V logic. The following figure shows how this is done.

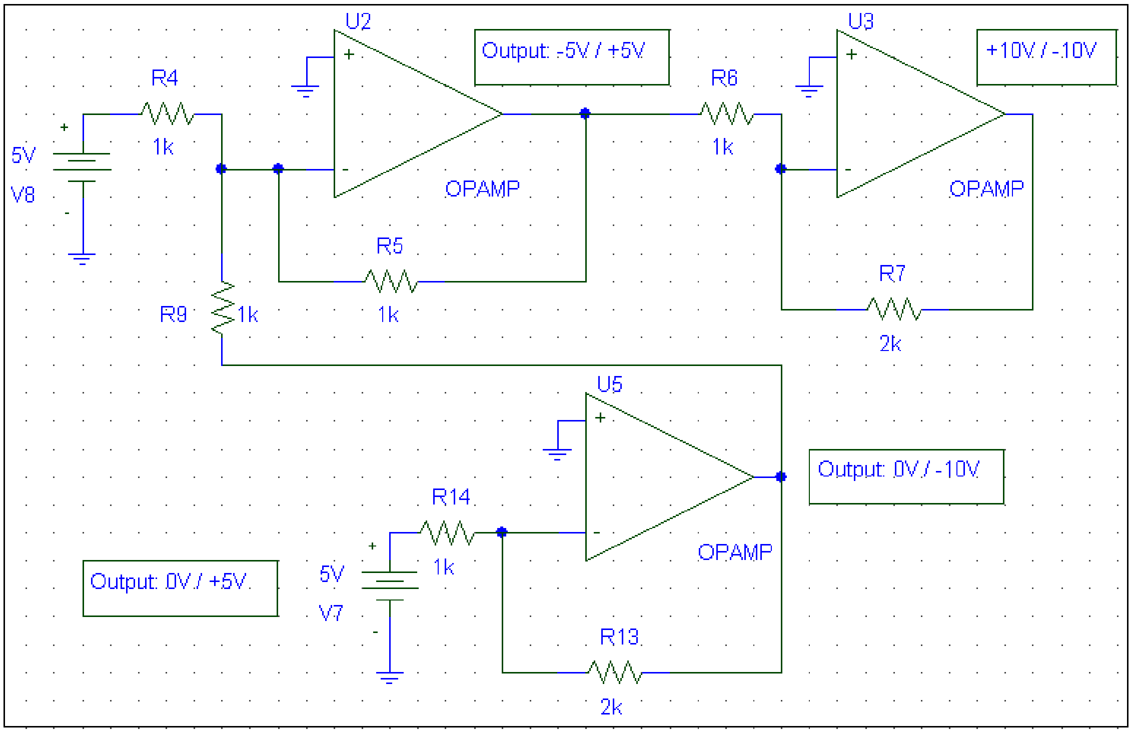


Fig 6.3 - Circuit Schematic to feed Transceiver RXD output to RS-232

Working: Following flow chart illustrates how the transceiver communication operates

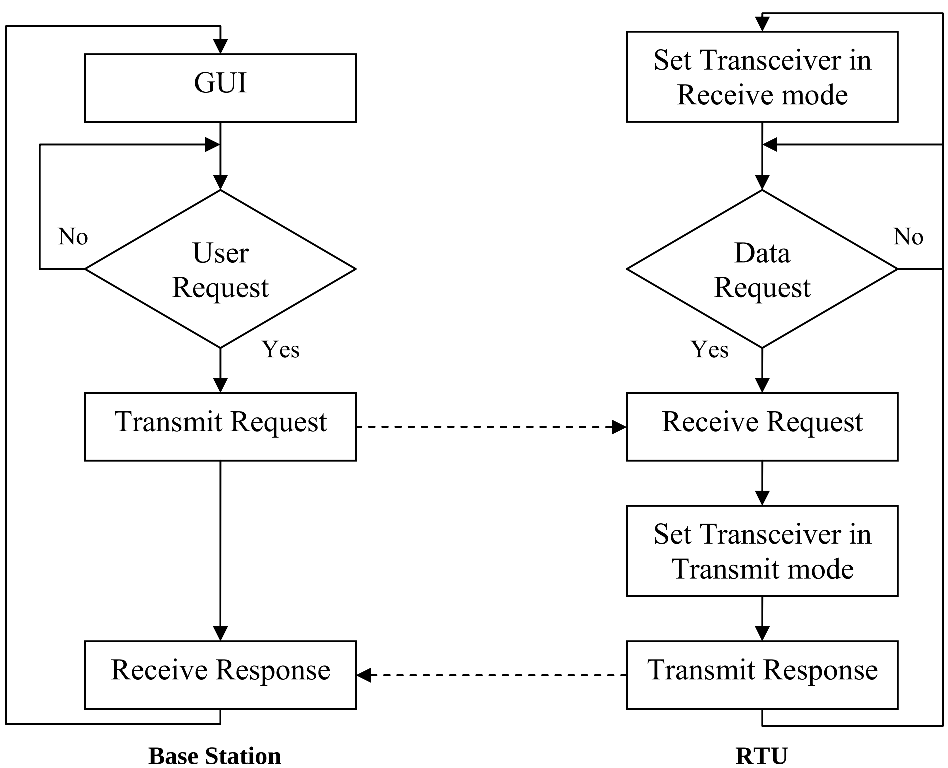


Fig 6.4 – Flowchart for wireless communication between Base Station and RTU

7. Cyclic Redundancy Check

7.1 Objective of CRC

Cyclic redundancy check or CRC is an error detection algorithm. It is used in the DNP 3.0 protocol. CRC algorithms are designed to maximize the probability of error detection. The probability that a message contains errors and the CRC stills checks out is very low. This procedure ensures the validity of the data received.

An (n+1) bit message is represented by a polynomial of degree n. Then using CRC algorithm k bits are computed from the (n+1) bit message. The additional k bits are sent over to the receiver which then again computes these k bits using the same algorithm. If the k bits computed matches with the k bits received by the receiver then there is no error, else error is considered to be detected.

7.2 Algorithm

Given a message polynomial M(x) of degree n, we select a divisor polynomial C(x) of degree k. Then our goal is to find a polynomial P(x) of degree (n + k) such that P(x) is exactly divisible by C(x). This is done as follows –

- Multiply M(x) with x^k to obtain T(x)
- Divide T(x) by C(x), obtain remainder as R(x)
- P(x) = T(x) – R(x)

Different divisor polynomials are available.

Example: Message – 11100101, Divisor $-C(x) = x^5 + x^4 + x + 1$, 1101 which is equivalent to 1101. Firstly, four zeros are appended at the end of the message. The resulting bit pattern is then divided by 1101. The remainder obtained is the CRC of the message. It is appended to the original message (without the zeros).

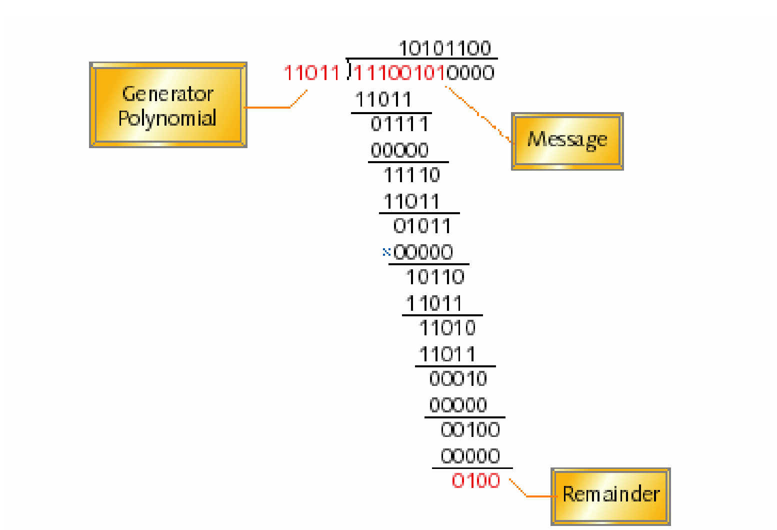


Fig 7.1 – Obtaining CRC for a ‘Message’ using a ‘Generator Polynomial’

7.3 Hardware Implementation

In practical applications, CRC is implemented by the use of registers as shown in the figure below. For hardware implementation, shift registers are used while memory allocation suffices for software implementation. The message bits are fed one by one starting with the most significant bit. After all the message bits have been fed, the resulting state of the registers gives the CRC of the message. The same implementation is used for verification of CRC at the receiver end. The incoming message is fed to the setup bit by bit. After the last bit is fed, the registers should all be 0.

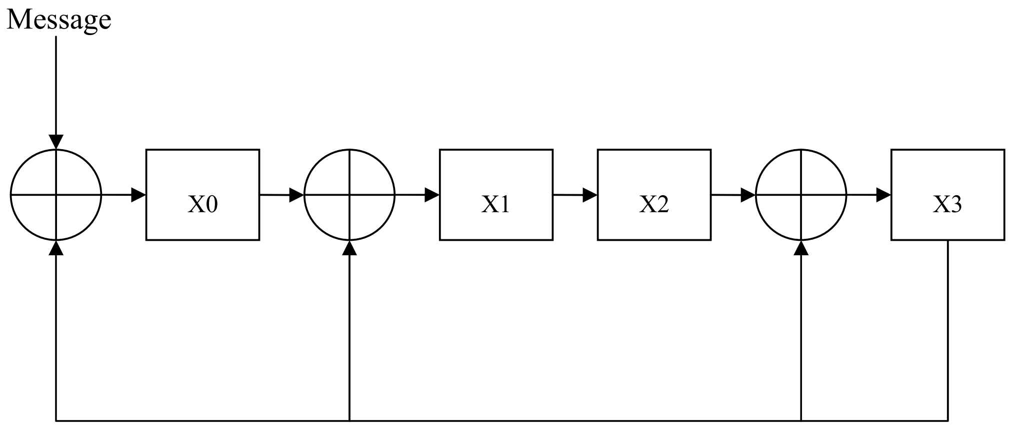


Fig 7.2 - Schematic for hardware implementation of CRC

Table 7.1

S. No.	X0	X1	X2	X3	Message Bit
1	0	0	0	0	1
2	1	0	0	0	1
3	1	1	0	0	1
4	1	1	1	0	0
5	0	1	1	1	0
6	1	1	1	0	1
7	1	1	1	1	0
8	1	0	1	0	1
9	1	1	0	1	0
10	1	0	1	1	0
11	1	0	0	0	0
12	0	1	0	0	0
CRC	**0**	**0**	**1**	**0**	

As seen from the table the CRC is obtained is 0100 (X3 – X0) which matches the value obtained from long division.

7.4 Software Implementation

Here instead of shift registers memory variables are used. Using the same example we compute the CRC as follows.

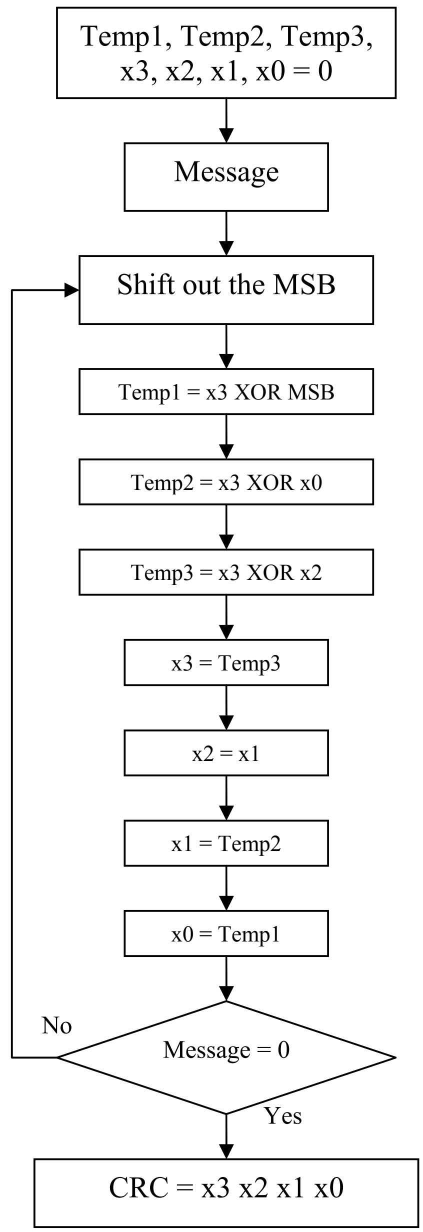


Fig 7.3 – Flowchart for software implementation of 4-bit CRC

The base station and the RTU implement 8–bit CRC using the polynomial $C(x) = x^8 + x^2 + x + 1$. The above methodology is extended for 8-bit as well.

8. Experimental Results and Discussions

This chapter gives details of the experimental setup and results obtained. The hardware schematic is shown in figure 8.1. The PC implements the graphical user interface (GUI) whereas the Intel 80196 microcontroller is mainly used for RTU computation.

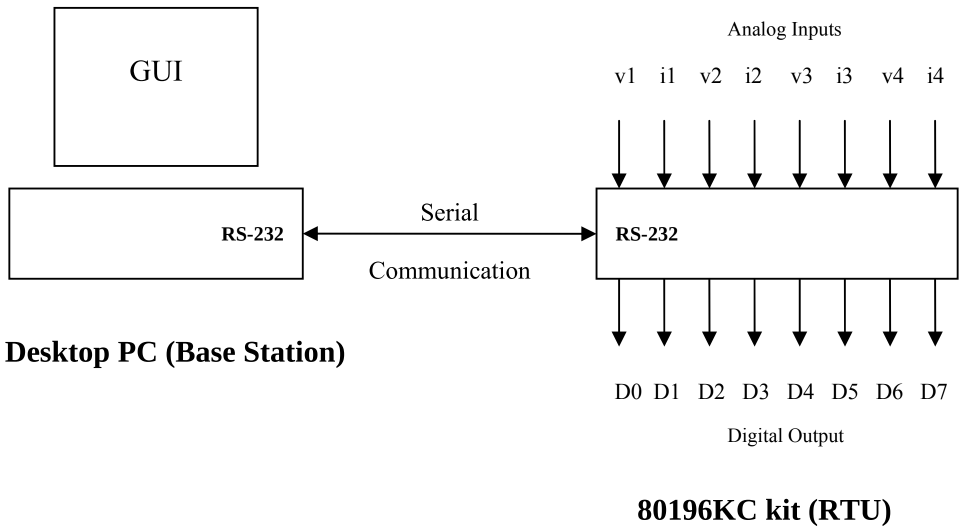


Fig 8.1 – Hardware schematic

8.1 Performance

To analyze system performance a signal S = (2.5 + 2.5*sinwt) was fed to channel 1 and 2 both. Offset was given so as to ensure that the input is in the range of 0-5 Volts. Observed values were –

Vrms = **3.03** Volt
Irms = **3.03** Volt
Average Power = **9.19** Watts
Power Factor = **0.9978**

Theoretical values can computed as follows -

Input, V = 2.5 + 2.5*sinwt
I = 2.5 + 2.5*sinwt

Expected values of Vrms and Irms are –

$$\text{Vrms} = \sqrt{\ \{ (1/2\pi) * \int_0^{2\pi} (2.5 + 2.5*\sin wt)^2\, dwt \}}$$

$$= \sqrt{\{(1/2\pi) * \int_0^{2\pi} (6.25 + 6.25*\sin^2 wt + 13.5*\sin wt)\, dwt\ \}}$$

$$= \sqrt{\{6.25 + 6.25/2)}$$

= **3.06** Volt

$$\text{Value of Average Power} = (1/2\pi) * \int_0^{2\pi} (2.5 + 2.5*\sin wt)^2\, dwt\ \}$$

$$= (1/2\pi) * \int_0^{2\pi} (6.25 + 6.25*\sin^2 wt + 13.5*\sin wt)\, dwt\ \}$$

$$= \{6.25 + 6.25/2)$$

= **9.375** Volt

Irms = Vrms
Power Factor = **1** (Since the V and I are just the same signal)

Error in observed values (for Vrms / Irms) = (3.06-3.03)*100/3.06 = **0.98 %**

Error in observed values (for Average Power) = (9.375-9.19)*100/9.375 = **1.97 %**

8.2 Sources of Error

1. *Quantization*: Observed values are based on 16 samples per cycle while the expected values are based on continuous spectrum. Improving sampling resolution will add more insensitivity to error.

2. *Square root*: Linear interpolation is applied while computing the square root. This approximation is negligibly erroneous only for very high numbers.

3. *Frequency mismatch*: The real time sampler assumes the input frequency to be 50 Hz but in reality there are always deviations from 50 Hz. Frequency variations will cause more or less than 16 samples to be taken in one cycle, thereby affecting RMS values drastically.

8.3 Screen Captures of the Graphical User Interface

Shown below is the Graphical User Interface for the digital output operation. The "Operate Line" command simply toggles the state of the selected line. That is if the line is on, it is switched off and vice versa

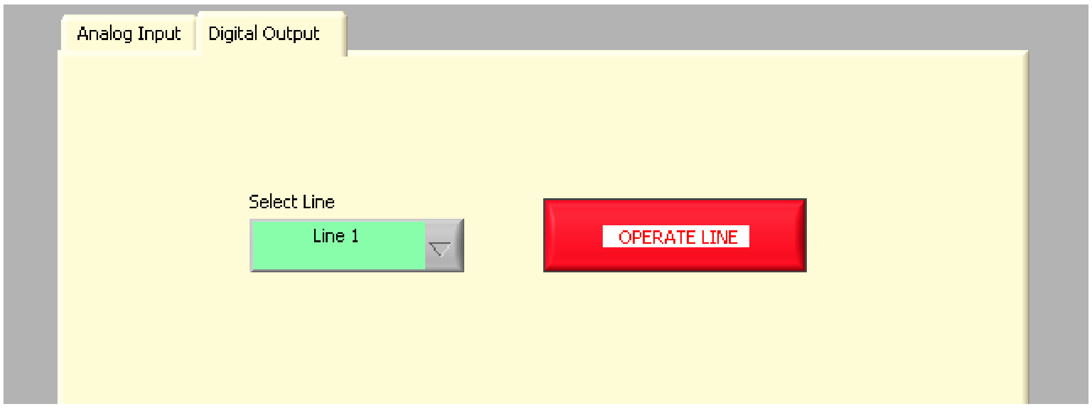


Fig 8.2 – Digital output control at Base Station

The base station can operate in two modes for data acquisition i.e. the continuous update mode and the discrete update mode. Shown below is the continuous update mode. The four plots represent RMS voltage, RMS current, average power and power factor for line 1. (All the voltages and currents have to be scaled down to 0-5V through a signal conditioning module, before they can be fed to the RTU). The string read gives the hexadecimal equivalent of the bit stream read by the base station. This contains 9 bytes of header, 2 bytes of data and 1 byte of CRC for the data.

Fig 8.3 - Continuous Update mode

Shown below is the discrete update mode at the Base Station. Data is only acquired upon clicking the update button. The four dials show the values of RMS voltage, RMS current, average power and power factor. The string read again gives the hexadecimal equivalent of the bit stream read by the base station.

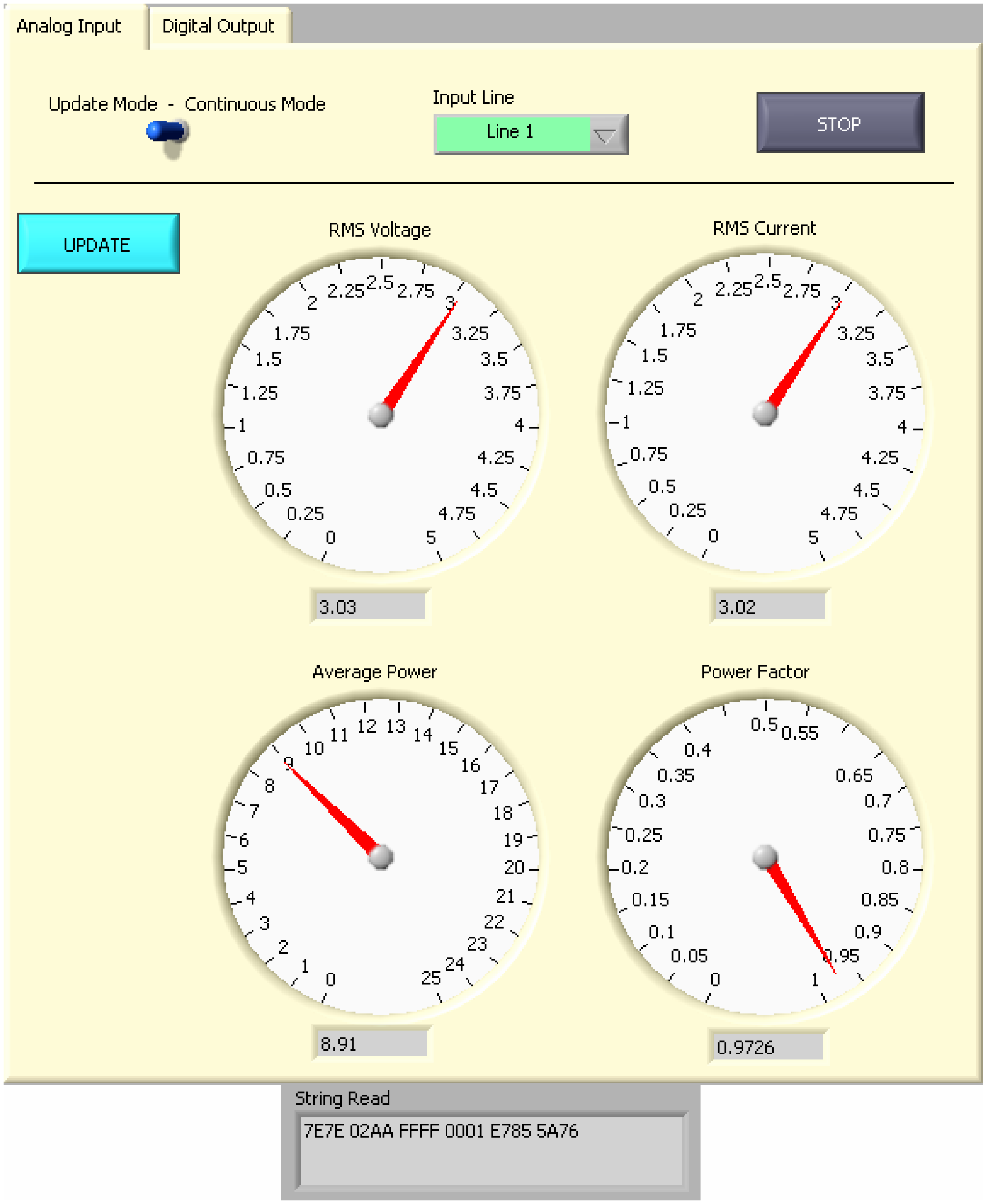
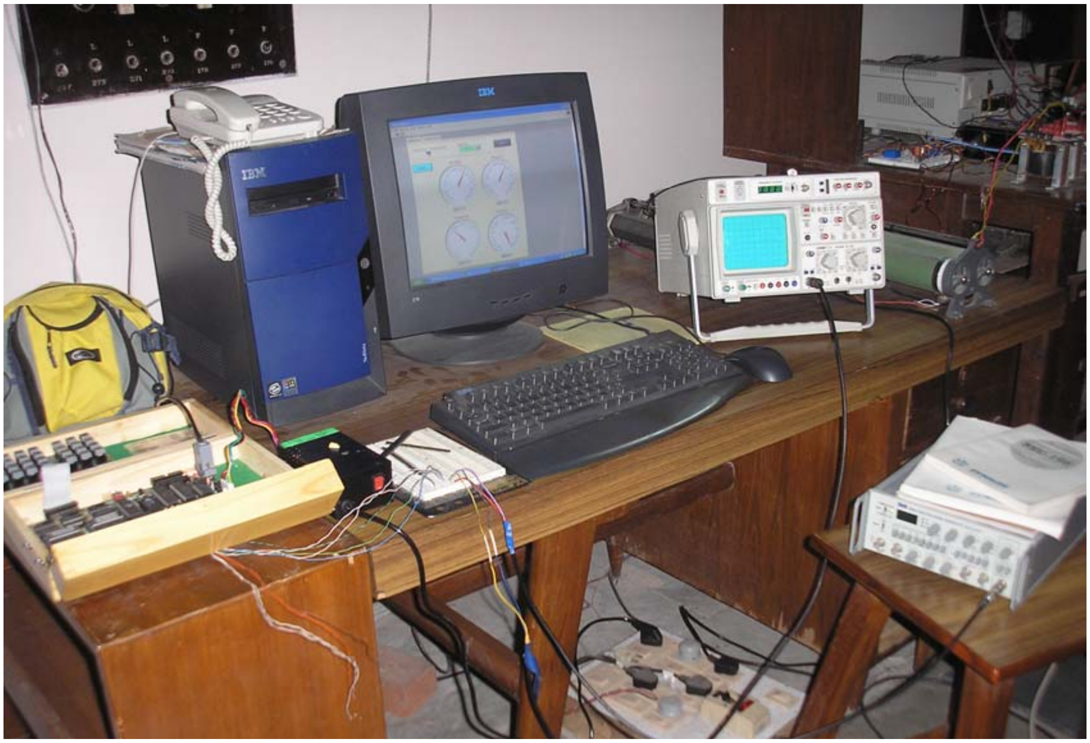
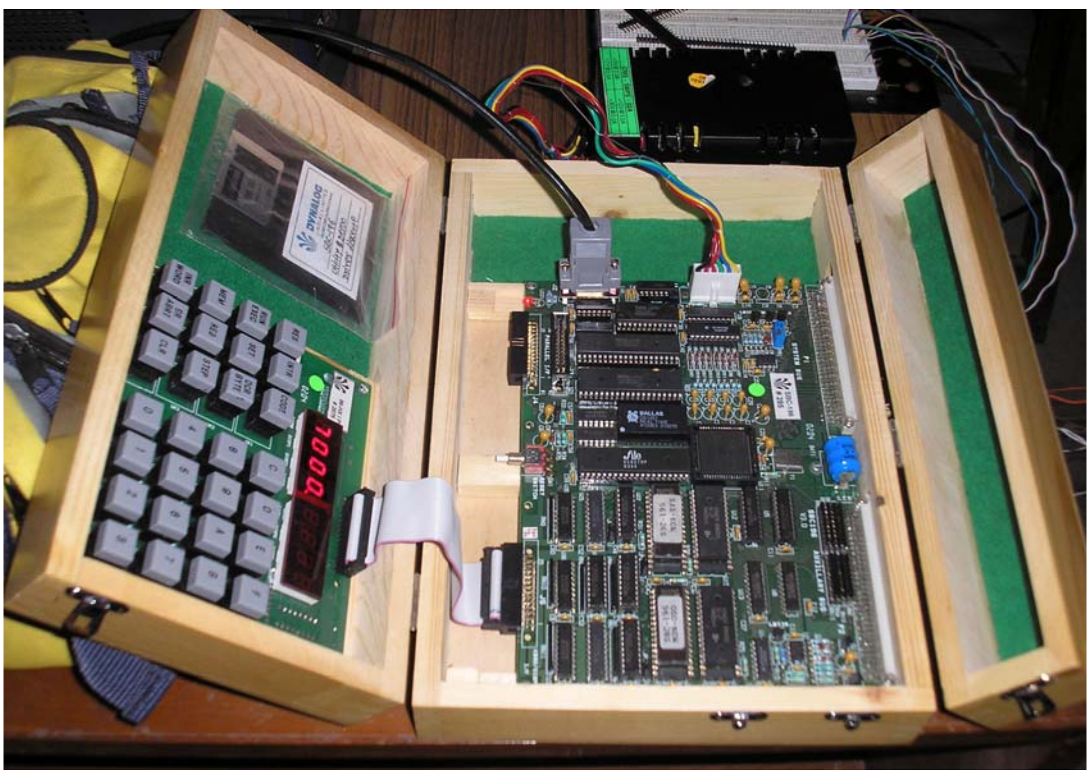


Fig 8.4 - GUI Screenshot

Photographs of the setup





9. Conclusion

The system with the Base Station, RTU and the Communication handler was setup successfully. RTU could sample 8 analog inputs and produce 8 digital outputs. RTU computes RMS and average values of sampled data. Base Station initiates communication from the RTU and retrieves data from it. This data is then displayed graphically in the graphical user interface (GUI). Base Station and RTU communicate via serial communication through RS-232 ports. DNP 3.0 protocol provides protection against noise and signal distortion. Cyclic Redundancy Checks provides communication reliability. Schematics for wireless communication with compatibility details are provided.

DNP3 causes transmission delay between Base Station and RTU to be around half a second. This is quite reasonable considering that it is not a very high speed application device. Percentage error of the computations performed by the RTU is 1.32, which is within tolerable limits.

Scope for future work –

1. In case of any emergency the RTU should be able to initiate communication with the Base Station. This requires analyzing the measured results at the RTU followed by communication setup.

2. RTU can be provided with external memory which could be used to store history of measured data. This could serve just as a black box in case of any damage to the RTU.

3. The kit used in this project is a general platform for development. We could make RTU more application specific by using a stand alone 80196 microcontroller.

10. References

[1] SBC-196 Technical Reference Manual – Dynalog (India) Limited

[2] SBC-196 User's Manual – Dynalog (India) Limited, October 2001, Rev. 1.0

[3] http://www.intel.com/design/mcs96/

[4] Virtual Instrumentation using LabVIEW – Sanjay Gupta and Joseph John, Tata McGraw-Hill, 2005 .

[5] http://www.dnp.org

[6] A DNP3 Protocol Primer – Ken Curtis, Woodland Engineering; Revision A, 20 March 2005.

[7] Computer Networks, A systems approach 3rd Edition – Larry L. Peterson and Bruce S. Davie, Morgan Kaufmann Publishers 2004.

Appendix A: Assembly Codes

Assembly Program Written with 80196KC Microcontroller

```
;******************************************************************
**
;
; 8096.INC - DEFINITION OF SYMBOLIC NAMES FOR THE I/O REGISTERS
OF
;       THE 8096 AND THE 80C196
;       (C) INTEL CORPORATION 1983
;******************************************************************
***
;
;/*
; *    8096 SFR's
; */
R0              EQU  00H:WORD    ; R   ZERO REGISTER
AD_COMMAND      EQU  02H:BYTE    ;  W
AD_RESULT       EQU  02H:WORD    ;
AD_LO           EQU  02H:BYTE    ; R
AD_HI           EQU  03H:BYTE    ; R
AD_TIME         EQU  03H:BYTE    ; W
HSI_MODE        EQU  03H:BYTE    ;  W
HSO_TIME        EQU  04H:WORD    ;  W
HSI_TIME        EQU  04H:WORD    ; R
PTSSEL          EQU  04H:WORD    ; W
HSO_COMMAND     EQU  06H:BYTE    ;  W
HSI_STATUS      EQU  06H:BYTE    ; R
PTSSRV          EQU  06H:WORD    ; W
SBUF            EQU  07H:BYTE    ; R/W
INT_MASK        EQU  08H:BYTE    ; R/W
INT_PEND        EQU  09H:BYTE    ; R/W
WATCHDOG        EQU  0AH:BYTE    ;  W  WATCHDOG TIMER
TIMER1          EQU  0AH:WORD    ; R
TIMER2          EQU  0CH:WORD    ; R
IOC3            EQU  0CH:BYTE    ; W
BAUDRATE        EQU  0EH:BYTE    ;  W
PORT0           EQU  0EH:BYTE    ; R
PORT1           EQU  0FH:BYTE    ; R/W
PORT2           EQU  10H:BYTE    ; R/W
SP_CON          EQU  11H:BYTE    ;  W
SP_STAT         EQU  11H:BYTE    ; R
IOC0            EQU  15H:BYTE    ;  W
IOS0            EQU  15H:BYTE    ; R
IOC1            EQU  16H:BYTE    ;  W
IOS1            EQU  16H:BYTE    ; R
PWM0_CONTROL    EQU  17H:BYTE    ;  W
PWM1_CONTROL    EQU  16H:BYTE    ;  W
PWM2_CONTROL    EQU  17H:BYTE    ;  W
```

```
PWM_CONTROL     EQU   17H:BYTE     ;  W
SP              EQU   18H:WORD     ; R/W
;
;    80C196 SFR's
IOC2            EQU   0BH:BYTE     ;  W
INT_PEND1       EQU   12H:BYTE     ; R/W
INT_MASK1       EQU   13H:BYTE     ; R/W
WSR             EQU   14H:BYTE     ; R/W
IOS2            EQU   17H:BYTE     ; R
T2CNTC          EQU   0CH:BYTE     ; R/W
```

```
; Scans and Operates on Analog Input

        include 80c196kc.inc

        rseg at 1Ah

ax:             dsw     1
bx:             dsw     1
cx:             dsw     1
dx:             dsw     1
ex:             dsw     1
fx:             dsw     1
temp:           dsw     1
disp_data:      dsw     1
disp_adrs:      dsw     1
disp_dcml:      dsb     1
cmd_mode:       dsb     1


square:         dsw     1
pointer:        dsw     1
iter:           dsw     1
accum:          dsw     1

counter:        dsb     1
rms:            dsw     1
rms_pointer:    dsw     1
rms_hist:       dsb     1
rms_recent:     dsw     1

samples:        dsb     1
destination:    dsw     1
channel:        dsb     1
temp1:          dsb     1
temp_2:         dsw     1
temp2:          dsw     2

loop_count:     dsb     1
inter1:         dsw     1
inter2:         dsw     1
disp_data_temp: dsw     1
disp_data_tmp1: dsw     1
table_ptr:      dsw     1
pointer_2:      dsw     1

power_acc1:     dsw     1
power_acc2:     dsw     1



;Registers for DNP 3.0 implementation
;------------------------------------

receive_flag:   dsb     1
send_flag:      dsb     1
synchronize:    dsb     1
my_add_L:       dsb     1
my_add_H:       dsb     1
master_add_L:   dsb     1
master_add_H:   dsb     1
ACK:            dsb     1
read_buffer:    dsb     1
transmit_buf1:  dsb     1
transmit_buf2:  dsb     1

CRCin           dsb     1
CRCout          dsb     1
CRCloop         dsb     1
x0              dsb     1
x1              dsb     1
x2              dsb     1
```

```
x3              dsb     1
x4              dsb     1
x5              dsb     1
x6              dsb     1
x7              dsb     1
tempIN          dsb     1
tempReg1        dsb     1
tempReg2        dsb     1
tempReg3        dsb     1


long_power      equ     power_acc1      :long
;...........................................................
spl             equ     sp      :byte
sph             equ     (sp+1)  :byte

long_bx         equ     bx      :long
al              equ     ax      :byte
ah              equ     (ax+1)  :byte
bl              equ     bx      :byte
;bh             equ     (bx+1)  :byte
cl              equ     cx      :byte
ch              equ     (cx+1)  :byte
dl              equ     dx      :byte
dh              equ     (dx+1)  :byte
el              equ     ex      :byte
eh              equ     (ex+1)  :byte
fl              equ     fx      :byte
fh              equ     (fx+1)  :byte
disp_dat0       equ     disp_data       :byte
disp_dat2       equ     (disp_data+1)   :byte
disp_adr0       equ     disp_adrs       :byte
disp_adr2       equ     (disp_adrs+1)  :byte
temp_l          equ     temp            :byte
temp_h          equ     (temp+1)        :byte


buf:            dsb     1
buf_data:       dsb     1
check1:         dsb     1
check2:         dsb     1
read_rg:        dsb     1
;................................................
;               KEY CODE DECLARATIONS

brk_key         equ     10h     :byte
;.................................................................
;library routines adrs

time_lib        equ     4002h           :word
print_lib       equ     4006h           :word
serial_lib      equ     400ah           :word
hex_lib         equ     400eh           :word

stack_lmt       equ     6200h           :word
user_stack      equ     6300h           :word

buffer          equ     6132h   :word   ;string
srl_sts_sav     equ     615Bh           :byte
;...........................................................

        cseg at 7000h

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                               ;
;       Program Starts Here     ;
;                               ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


        ;
```

```
        ;;      This Slave's Master's Address is ffff h
        ;;;     This Slave's address is 0001 h
        ;;      Synchronize byte recognized by this slave is 01111110 b
        ;

        ldb     synchronize,#01111110b  ;synchronize byte
        ldb     my_add_H,#00h           ;higher byte of slave address
        ldb     my_add_L,#01h           ;lower byte of salve address
        ldb     master_add_H,#0ffh      ;higher byte of master address
        ldb     master_add_L,#0ffh      ;lower byte of master address
        ldb     ACK,#10101010b          ;ACK byte for link control




        ;
        ;; Initialize flags used for DNP communication
        ;

        ldb     receive_flag,#00h
        ldb     send_flag,#00h


        ld      sp,#6400h
        ;ld     bx,#msg
        ;ldb    el,#3
        ;lcall  serial_lib



        ;
        ;; This call creates square root look-up table
        ;; Once created this call can be commented
        ;

        ;lcall  sqrt_table



        ;
        ;; Setting pointers for data storage
        ;

        ld      rms_pointer,#9000h
        ldb     rms_hist,#05h
        ld      rms_recent,#8FE8h



        ;
        ;;  settings for serial communication
        ;;;     Baud Rate = 2400
        ;;      Clock Source = XTAL1
        ;

        ldb     BAUDRATE,#39h
        ldb     BAUDRATE,#81h
        ldb     SP_CON,#1bh
        ldb     IOC1,#20h
        ld      fx,#600ch
        ld      bx,#ser_req
        st      bx,[fx]
        ldb     read_rg,#00h



start:  ;
        ;; Start sampling - take 16 samples per cycle for 4 cycles i.e. 64 samples
        ;

        ldb     samples,#40h            ;No of samples to acquire
```

```
        ld      destination,#8000h      ;starting address of stored values
        ;
        ;; Timer settings
        ;
        ldb     ioc2,#40h               ;count every 8 machine states, count up, enable comman
        ldb     ioc0,#02h               ;reset timer2
        ;
        ;; HSO unit settings - sampling frequency of 16 samples/cycle
        ;
        ldb     HSO_COMMAND,#11001111b  ;CAM lock, start AD conversion based on timer2
        ld      HSO_TIME,#03aah         ;938d sampling interval (shd be 03aah)
        ld      fx,#0000h
        ;
        ;; Interrupt Settings
        ;
        ld      bx,#6002h
        ld      ex,#rd_adc
        st      ex,[bx]                 ;write address of interupt service subroutine
        ldb     channel,#10h
        ldb     ad_command,channel      ;start conversion by hso
        ldb     int_mask,#42h           ;enable AD complete interrupt
        ei
        ;
        ;; Reset timer - sampling frequency of 16 samples/cycle
        ;
        ldb     HSO_COMMAND,#11001110b  ;CAM lock, reset timer2
        ld      HSO_TIME,#03abh         ;time to reset timer2
        pusha
        ldb     WSR,#01h                ;switch to horizontal window 1
        ldb     T2CNTC,#01h             ;clock internally
        popa
Wait:   ;
        ;; Wait for 16 samples to be taken
        ;
        jne     wait
        incb    channel                 ;scan next channel
        ldb     samples,#40h
        inc     destination
        cmpb    channel,#18h
        je      fin_samp
        ldb     ad_command,channel
        sjmp    wait                    ;again wait for next 16 samples
fin_samp:       ;
                ;; all 8 channels sampled; proceed to calculations
                ;
```

```

                ldb     ioc2,#80h               ;clear CAM




                ;
                ;;   8 channels have 8 RMS values and 4 Pavg values
                ;;;  Both RMS and Avg values yield 2 bytes of data
                ;;;; one time sampling of all 8 channels yield 24 bytes of data
                ;;;  we store 4 time sampling result
                ;;   In the 5th time we loop-back i.e. overwrite the 1st time result
                ;

                cmp     rms_recent,#9060h       ; Is it the 5th time
                jne     no_loopback

                ld      rms_recent,#8fe8h       ; Yes, then overwrite the 1st time result
no_loopback:    add     rms_recent,#18h         ; No, simply store the result in the next loca



        ;
        ;; Compute the RMS values of the 8 channels
        ;

loop:   ld      pointer,#8000h
        ldb     cl,#08h

cmpt:   ldb     counter,#40h
        lcall   r_m_sq
        st      rms,[rms_pointer]

        inc     rms_pointer
        inc     rms_pointer
        djnz    cl,cmpt



        ;
        ;; Compute Pavg = Summation(Vi X Ii)/ n
        ;; Assume ch0 to be voltage,ch1 to be current and so on
        ;

power:  ld      pointer,#8000h
        ld      pointer_2,#8040h
        ldb     loop_count,#04h

n_ch_p: ldb     counter,#40h
        ld      bx,#0000h
        ld      cx,#0000h
        ld      power_acc1,#0000h
        ld      power_acc2,#0000h
        ;ld     cx,#0000h

avg_pw: ldb     fl,[pointer]
        ldb     fh,[pointer_2]
        mulub   temp,fl,fh

        add     power_acc1,temp
        addc    power_acc2,cx

        inc     pointer
        inc     pointer_2
        djnz    counter,avg_pw

        shr     power_acc1,#06h
        shl     power_acc2,#0ah
        add     power_acc1,power_acc2

        ;shrl   long_power,#06h
```

```
        st      power_acc1,[rms_pointer]

        inc     rms_pointer
        inc     rms_pointer

        ;ld     disp_data,power_acc1
        ;lcall  disp

        add     pointer,#0040h
        add     pointer_2,#0040h
        djnz    loop_count,n_ch_p


;enable serial port interrupt here

        ;ldb    INT_MASK,#40h




        ;
        ;; all 8 channels and computed once
        ;; Repeat the entire process again
        ;

next:           djnz    rms_hist, n_sample
                ld      rms_pointer,#9000h
                ldb     rms_hist,#05h
n_sample:       ljmp    start                   ;do next sampling

idle:   sjmp    idle                            ;never reached



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                               ;
;       Main program ends       ;
;                               ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;



;---------------------------------------------------------------------


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                                       ;
;       Sub routines start here         ;
;                                       ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;
;;
;  Sub routine to compute root mean square value
;       @arg    pointer - points to first sample
;       @arg    counter - no of samples to compute rms of
;       @return rms - contains the root mean square value

r_m_sq: ld      square,#0000h
sum_sq: ldb     fl,[pointer]

        ldb     fh,fl
        mulub   temp,fh,fl
        shr     temp,#06h
        add     square,temp             ;'square' stores the sum of squares of acquired data
        inc     pointer
        djnz    counter,sum_sq

        cmp     square,#0000h
        jh      intplt
        ld      rms,#0000h
```

```
        ret

intplt: lcall   traverse
        sub     temp2,square,inter1
        ldb     temp1,#10h
        shll    temp2,temp1
        sub     temp,inter2,inter1
        divu    temp2,temp
        ld      dx,temp2
        ldb     fh,fl
        ldb     fl,dh

        cmpb    fl,#00h
        jne     no_cor
        incb    fh
no_cor: ld      rms,fx
        ret


;-------------------------------------------------------------------

;;;
;;
;  Sub Routine to travers the square root table
;       @return fl - holds the floor sqrt integer
;       @return fh - holds the ceil sqrt integer
;       @return inter1 - square of fl
;       @return inter2 - square of fh

traverse:       ld      table_ptr,#7500h
                ldb     al,#00h

search:         inc     table_ptr
                inc     table_ptr
                incb    al
                ld      temp,[table_ptr]
                cmp     temp,square
                je      found
                jh      found
                sjmp    search

found:          ld      inter2,[table_ptr]
                dec     table_ptr
                dec     table_ptr
                ld      inter1,[table_ptr]

                ldb     fh,al
                subb    fl,al,#01h
                ret

;---------------------------------------------------------------------

;;;
;;
;  Sub routine for a/d conversion complete event

rd_adc: ldb     fh,ad_hi                ;load low order byte
        stb     fh,[destination]        ;storing sampled data to memory

        djnz    samples,cont
        cmpb    samples,#00h
        sjmp    finish

cont:   inc     destination             ;increament destination address
        ldb     ad_command,channel      ;start conversion by hso

finish: ret


;--------------------------------------------------------------------
```

```
;;;
;;
;  Subroutine to handle serial interrupt

ser_req:        pusha

                ;Reset sampling
                ldb     samples,#40h
                ld      destination,#8000h
                ldb     channel,#10h

                ;Check for RI/TI interrupt
                ldb     buf,SP_STAT
                andb    check1,buf,#40h
                jne     ri
                andb    check2,buf,#20h
                jne     ti
                sjmp    ser_fin

ri:             lcall   ri_req
                sjmp    ser_fin
ti:             lcall   ti_req

ser_fin:        popa
                ret


;-----------------------------------------------------------------------

;;;
;;
;  Subroutine to handle RI interrupt

ri_req:         ldb     buf_data,SBUF

                ;xorb   port1,#00000001b


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                             ;;
;;  This part of the code strips off the DNP 3.0 header from the received data  ;;
;;                                                                             ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;
;; For each byte of DNP header we assign a flag number in receive_flag
;; This flag number is used to distinguish a particular byte of DNP header
;



sync1_recv:     cmpb    receive_flag,#00000000b
                jgt     sync2_recv

                ;
                ;; Check the first synchronize byte
                ;
                cmpb    buf_data,synchronize
                jne     clear_recv1
                incb    receive_flag

                call    initializeCRC                   ; compute CRC for DNP header
                ldb     CRCin,buf_data
                call    computeCRC
                sjmp    fin_write


sync2_recv:     cmpb    receive_flag,#00000001b
                jgt     length_recv

                ;
```

```
                ;; Check the second synchronize byte
                ;
                cmpb    buf_data,synchronize
                jne     clear_recv1
                incb    receive_flag

                ldb     CRCin,buf_data                  ; compute CRC for DNP header
                call    computeCRC
                sjmp    fin_write


length_recv:    cmpb    receive_flag,#00000010b
                jgt     link_recv

                ;
                ;; Store remaining frame length (to be received)
                ;
                ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

                incb    receive_flag

                ldb     CRCin,buf_data                  ; compute CRC for DNP header
                call    computeCRC
                sjmp    fin_write


link_recv:      cmpb    receive_flag,#00000011b
                jgt     dest1_recv

                ;
                ;; Check the link control
                ;
                ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                ;jump to clear_recv if check fails

                incb    receive_flag

                ldb     CRCin,buf_data                  ; compute CRC for DNP header
                call    computeCRC
                sjmp    fin_write


dest1_recv:     cmpb    receive_flag,#00000100b
                jgt     dest2_recv

                ;
                ;; Check the upper byte of destination address
                ;
                cmpb    buf_data,my_add_H
                jne     clear_recv1
                incb    receive_flag

                ldb     CRCin,buf_data                  ; compute CRC for DNP header
                call    computeCRC
                sjmp    fin_write


clear_recv1:    sjmp    clear_recv


dest2_recv:     cmpb    receive_flag,#00000101b
                jgt     src1_recv

                ;
                ;; Check the lower byte of destination address
                ;
                cmpb    buf_data,my_add_L
                jne     clear_recv
                incb    receive_flag

                ldb     CRCin,buf_data                  ; compute CRC for DNP header
```

```
                call    computeCRC
                sjmp    fin_write


src1_recv:      cmpb    receive_flag,#00000110b
                jgt     src2_recv

                ;
                ;; Check the upper byte of source address
                ;
                cmpb    buf_data,master_add_H
                jne     clear_recv
                incb    receive_flag

                ldb     CRCin,buf_data                  ; compute CRC for DNP header
                call    computeCRC
                sjmp    fin_write


src2_recv:      cmpb    receive_flag,#00000111b
                jgt     crc_recv

                ;
                ;; Check the lower byte of source address
                ;
                cmpb    buf_data,master_add_L
                jne     clear_recv
                incb    receive_flag

                ldb     CRCin,buf_data                  ; compute CRC for DNP header
                call    computeCRC
                sjmp    fin_write


crc_recv:       cmpb    receive_flag,#00001000b
                jgt     data_recv

                ;
                ;; Check the CRC
                ;
                ldb     CRCin,#00h
                call    computeCRC
                call    resultCRC

                ;compare buf_data with CRCout
                ;jump to clear_recv if check fails
                cmpb    buf_data,CRCout
                jne     clear_recv
                incb    receive_flag
                sjmp    fin_write


data_recv:      cmpb    receive_flag,#00001001b
                jgt     dataCRC_recv
                ldb     read_buffer,buf_data
                incb    receive_flag
                sjmp    fin_write


dataCRC_recv:   ; match CRC byte received with that computed on read_buffer
                ; if successful then process the request i.e. jump to check_io
                ; else jump to clear_recv

                call    initializeCRC
                ldb     CRCin,read_buffer
                call    computeCRC
                ldb     CRCin,#00h
                call    computeCRC
                call    resultCRC

                ;compare buf_data with CRCout
```

```
                ;if comparision true => proceed else jump to check_io
                cmpb    buf_data,CRCout
                jne     clear_recv

                ldb     buf_data,read_buffer
                ldb     receive_flag,#00h
                sjmp    check_io


clear_recv:     ;
                ;; One or more of the DNP checks failed => Data corrupted
                ;; Sender has to re-transmit => receive_flag set to 00h
                ;

                ldb     receive_flag,#00h
                sjmp    fin_write               ;exit RI interrupt handler


;..............................................................................

check_io:       ;
                ;; To check for rms/avg value read or digital output
                ;
                andb    read_rg,buf_data,#40h
                jne     read_req

write_req:      ;
                ;; Read values
                ;
                ldb     ah,#00h
                andb    read_rg,buf_data,#20h
                je      write_rms
                ldb     ah,#08h

write_rms:      ld      fx,rms_recent
                andb    al,buf_data,#1fh
                addb    al,ah

ser_loop:       je      transmit
                inc     fx
                inc     fx
                decb    al
                sjmp    ser_loop

transmit:       ;
                ;; Trnsmit the byte read to the base station
                ;

                ldb     transmit_buf1,[fx]
                inc     fx
                ldb     transmit_buf2,[fx]
                ldb     SBUF,synchronize

                ;
                ;; Start computing CRC for header
                ;
                call    initializeCRC
                ldb     CRCin,synchronize
                call    computeCRC

                sjmp    fin_write



read_req:       ;
                ;; Digital output
                ;
                ldb     read_rg,#01h
                andb    fl,buf_data,#3fh
                shlb    read_rg,fl
                xorb    port1,read_rg
```

```

fin_write:      ret


;---------------------------------------------------------------------

;;;
;;
;  Suroutine to handle TI interrupt

ti_req:         ;
                ;; This subroutine gets called when one byte-transmission completes
                ;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                      ;;
;;  This part of the code appends DNP 3.0 header to the outgoing data   ;;
;;                                                                      ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;
;; For each byte of DNP header sent we assign a flag number in send_flag
;; This flag number is used to distinguish a particular byte of DNP header to be sent
;


sync2_send:     cmpb    send_flag,#00000000b
                jgt     length_send

                ;
                ;; Send Synchronize byte 2
                ;
                ldb     SBUF,synchronize
                incb    send_flag

                ldb     CRCin,synchronize       ; next byte to be computed CRC on
                call    computeCRC
                sjmp    fin_ti


length_send:    cmpb    send_flag,#00000001b
                jgt     link_send

                ;
                ;; Send the number of data bytes which follow the header
                ;
                ldb     SBUF,#02h

                ldb     CRCin,#02h              ; next byte to be computed CRC on
                call    computeCRC
                incb    send_flag
                sjmp    fin_ti


link_send:      cmpb    send_flag,#00000010b
                jgt     dest1_send

                ;
                ;; Link control
                ;
                ldb     SBUF,ACK

                ldb     CRCin,ACK               ; next byte to be computed CRC on
                call    computeCRC
                incb    send_flag
                sjmp    fin_ti


dest1_send:     cmpb    send_flag,#00000011b
                jgt     dest2_send

```

```
                ;
                ;; Send upper byte of Destination address
                ;
                ldb     SBUF,master_add_H

                ldb     CRCin,master_add_H      ; next byte to be computed CRC on
                call    computeCRC
                incb    send_flag
                sjmp    fin_ti


dest2_send:     cmpb    send_flag,#00000100b
                jgt     src1_send

                ;
                ;; Send lower byte of Destination address
                ;
                ldb     SBUF,master_add_L

                ldb     CRCin,master_add_L      ; next byte to be computed CRC on
                call    computeCRC
                incb    send_flag
                sjmp    fin_ti


src1_send:      cmpb    send_flag,#00000101b
                jgt     src2_send

                ;
                ;; Send upper byte of source address
                ;
                ldb     SBUF,my_add_H

                ldb     CRCin,my_add_H          ; next byte to be computed CRC on
                call    computeCRC
                incb    send_flag
                sjmp    fin_ti


src2_send:      cmpb    send_flag,#00000110b
                jgt     crc_send

                ;
                ;; Send lower byte of source address
                ;
                ldb     SBUF,my_add_L

                ldb     CRCin,my_add_L          ; next byte to be computed CRC on
                call    computeCRC
                incb    send_flag
                sjmp    fin_ti


crc_send:       cmpb    send_flag,#00000111b
                jgt     data_send1

                ;
                ;; CRC check
                ;
                ldb     CRCin,#00h
                call    computeCRC
                call    resultCRC

                ldb     SBUF,CRCout
                incb    send_flag
                sjmp    fin_ti


data_send1:     cmpb    send_flag,#00001000b
                jgt     data_send2
                ldb     SBUF,transmit_buf1
```

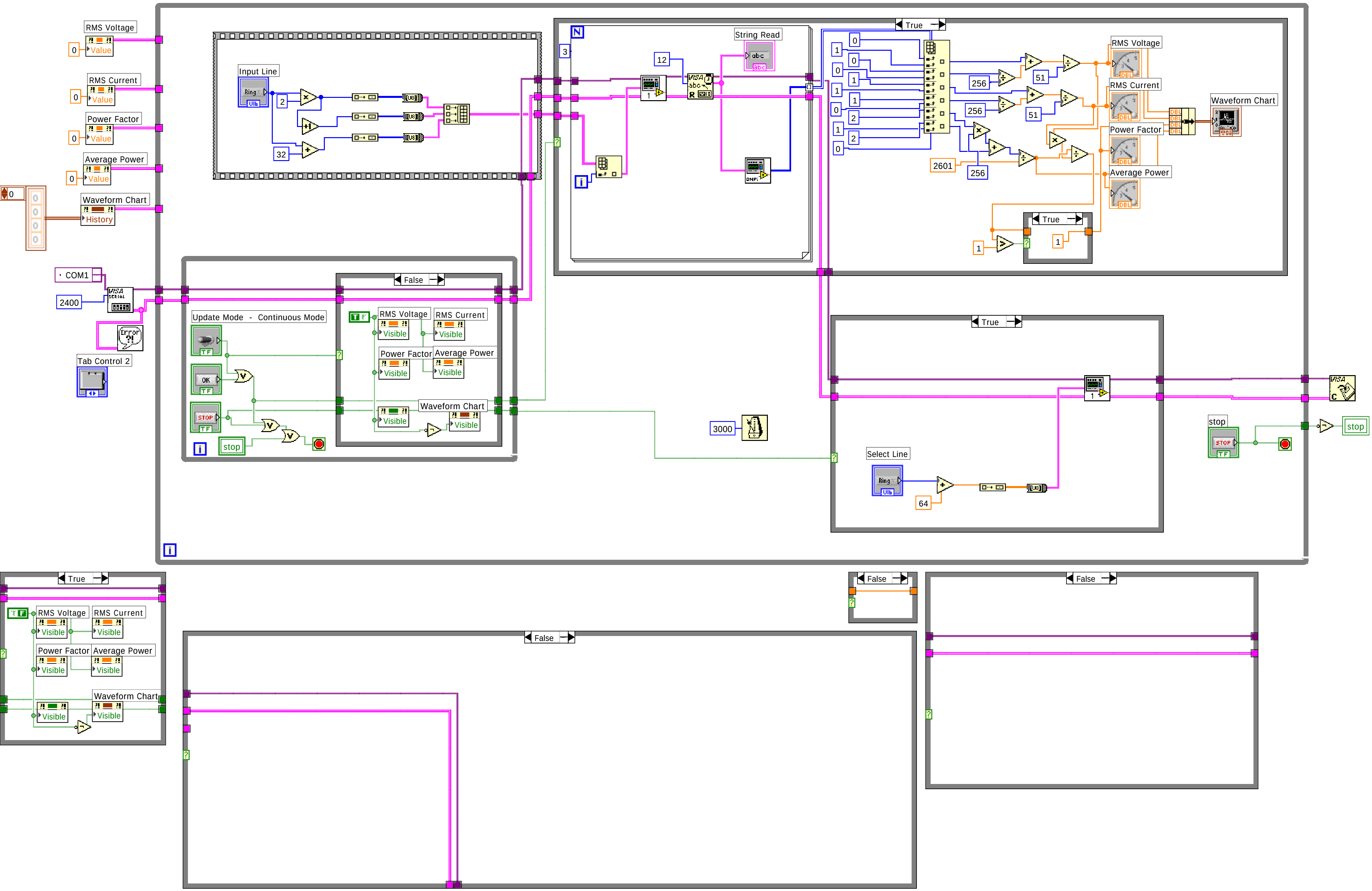
```
                incb    send_flag
                sjmp    fin_ti


data_send2:     cmpb    send_flag,#00001001b
                jgt     dataCRC_send
                ldb     SBUF,transmit_buf2
                incb    send_flag
                sjmp    fin_ti


dataCRC_send:   cmpb    send_flag,#00001010b
                jgt     fin_transmit

                ;
                ;; CRC check
                ;
                call    initializeCRC
                ldb     CRCin,transmit_buf1
                call    computeCRC
                ldb     CRCin,transmit_buf2
                call    computeCRC
                ldb     CRCin,#00h
                call    computeCRC
                call    resultCRC

                ldb     SBUF,CRCout
                incb    send_flag
                sjmp    fin_ti


fin_transmit:   ;
                ;; Full packet transmitted
                ;; Next packet transmission should from 1st byte of DNP header => send_flag = (
                ;

                ldb     send_flag,#00h

fin_ti:         ret


;--------------------------------------------------------------------------

;;;
;;
;  Subroutines to compute 8 bit CRC (Cyclic Redundancy Check) of one byte
;       @param CRCin - byte whose CRC has to be computed
;       @return CRCout - one byte CRC

initializeCRC:  ldb     x0,#00h
                ldb     x1,#00h
                ldb     x2,#00h
                ldb     x3,#00h
                ldb     x4,#00h
                ldb     x5,#00h
                ldb     x6,#00h
                ldb     x7,#00h

                ret



computeCRC:     ldb     CRCloop,#08h

loopCRC:        ldb     tempIN,#00h
                shlb    CRCin,#01h
                addcb   tempIN,tempIN
                ldb     tempReg1,x7
                xorb    tempReg1,x1
                ldb     tempReg2,x7
                xorb    tempReg2,x0
```

```
                ldb     tempReg3,x7
                xorb    tempReg3,tempIN

                ldb     x7,x6
                ldb     x6,x5
                ldb     x5,x4
                ldb     x4,x3
                ldb     x3,x2
                ldb     x2,tempReg1
                ldb     x1,tempReg2
                ldb     x0,tempReg3

                decb    CRCloop
                jne     loopCRC

                ret



resultCRC:      shlb    x7,#07h
                shlb    x6,#06h
                shlb    x5,#05h
                shlb    x4,#04h
                shlb    x3,#03h
                shlb    x2,#02h
                shlb    x1,#01h

                ldb     CRCout,#00h
                addb    CRCout,x0
                addb    CRCout,x1
                addb    CRCout,x2
                addb    CRCout,x3
                addb    CRCout,x4
                addb    CRCout,x5
                addb    CRCout,x6
                addb    CRCout,x7

                ret



;-------------------------------------------------------------------------


;;;
;;
;  This is a Sub routine to build a lookup table to compute square root

sqrt_table:     ld      fx,#7500h
                ldb     al,#00h
                ld      bx,#0000h
                st      bx,[fx]

store:          incb    al
                inc     fx
                inc     fx
                mulub   bx,al,al
                st      bx,[fx]
                cmpb    al,#0ffh
                je      done
                sjmp    store
done:           ret



;-------------------------------------------------------------------------

;;;
;;
;  This sub routine is for displaying a word to the terminal

```

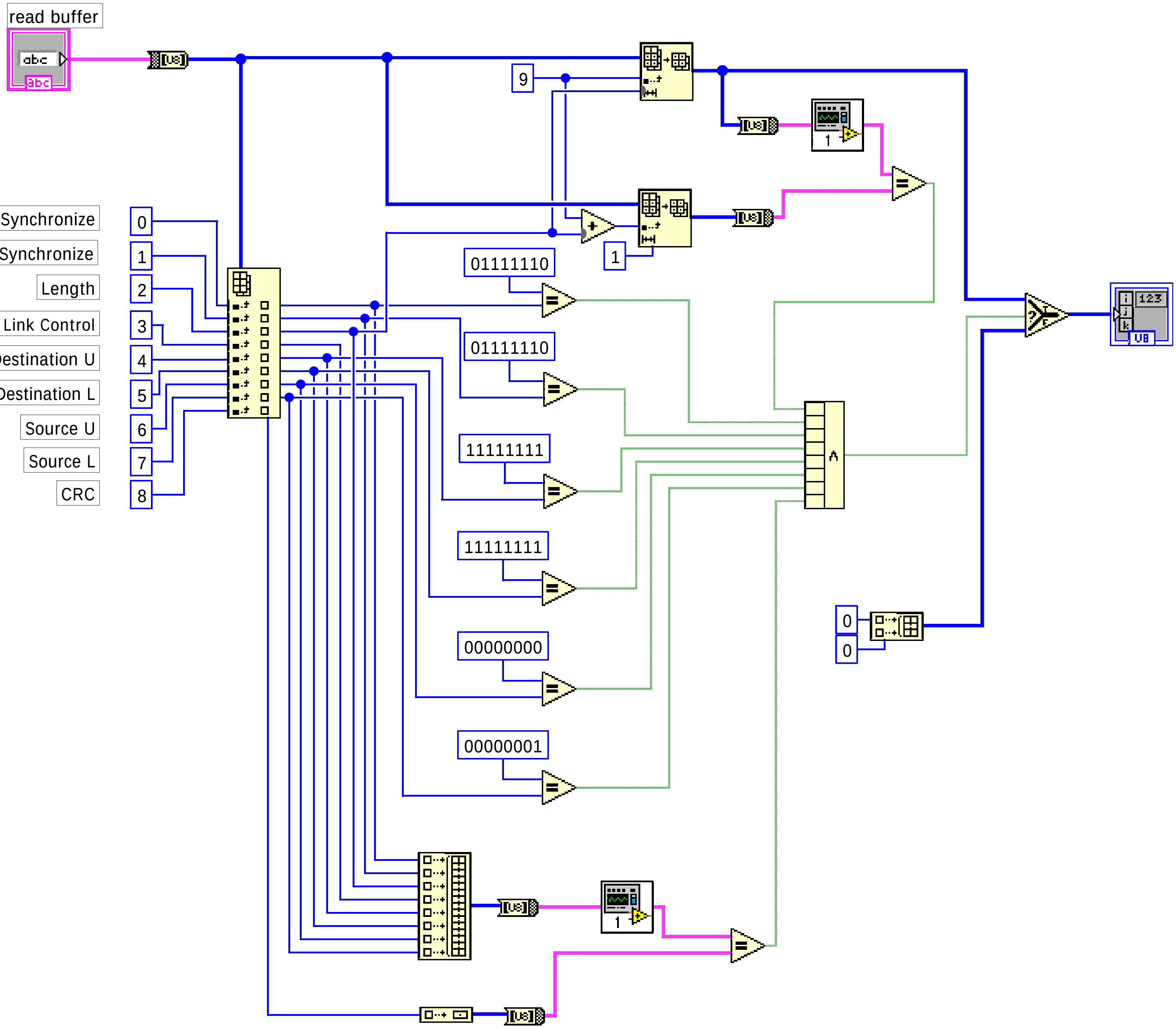
```
disp:   ld      disp_data_temp,ax
        ld      disp_data_tmp1,fx
        ld      ax,disp_data
        ldb     el,#06h
        lcall   serial_lib
        ld      fx,disp_data_tmp1
        ld      ax,disp_data_temp
        ret

 end
```

Appendix B: LabVIEW 7.1 Codes (Written with a PC)

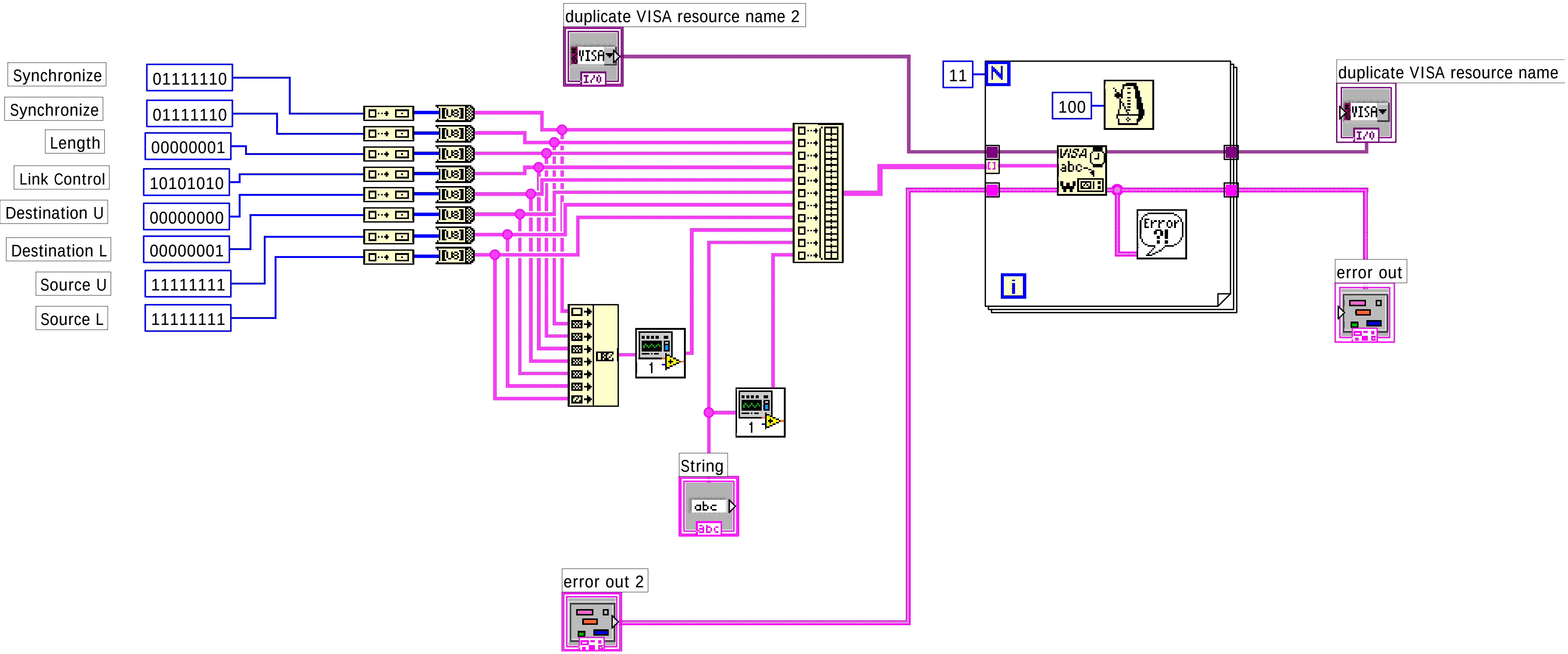
sem2_1.vi
D:\personal\courses\BTP\Sem2 Codes\sem2_1.vi
Last modified on 4/4/2006 at 1:09 PM
Printed on 4/16/2006 at 11:12 PM



DNP_in_2.vi
D:\personal\courses\BTP\Sem2 Codes\DNP_in_2.vi
Last modified on 4/4/2006 at 11:50 AM
Printed on 4/16/2006 at 11:09 PM



DNP_out_2.vi
D:\personal\courses\BTP\Sem2 Codes\DNP_out_2.vi
Last modified on 4/4/2006 at 1:09 PM
Printed on 4/16/2006 at 11:07 PM



CRC_1.vi
D:\personal\courses\BTP\Sem2 Codes\CRC_1.vi
Last modified on 4/4/2006 at 1:14 PM
Printed on 4/16/2006 at 11:10 PM