Universal "Chip Based Smart Card" Snooping Device

By

Dimitri Denamany
(EE 1467)

FINAL PROJECT DISSERTATION

Dissertation Submitted in partial fulfillment of
the requirements for the
Bachelor of Engineering (Hons)
(Electrical & Electronics Engineering)

JUNE 2004

Universiti Teknologi PETRONAS
Bandar Seri Iskandar
31750 Tronoh
Perak Darul Ridzuan

CERTIFICATION OF APPROVAL

Universal "Chip Based Smart Card" Snooping Device

by

Dimitri Denamany

A project dissertation submitted to the
Electrical Engineering Programme
Universiti Teknologi PETRONAS
in partial fulfilment of the requirement for the
BACHELOR OF ENGINEERING (Hons)
(ELECTRIC & ELECTRONICS ENGINEERING)

Approved by,

(Mr. Patrick Sebastian)

UNIVERSITI TEKNOLOGI PETRONAS

TRONOH, PERAK

JUNE 2004

CERTIFICATION OF ORIGINALITY

This is to certify that I am responsible for the work submitted in this project, that the original work is my own except as specified in the references and acknowledgements, and that the original work contained herein have not been undertaken or done by unspecified sources or persons.

DIMITRI DENAMANY

TABLE OF CONTENTS

LIST OF APPENDICES

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

The objective of this project is basically to design, build and test a **Universal "Chip Based Smart Card" Snooping Device.** This device would function to notify its user of the actual communication between the smart card and the smart card reader. The existence of this device not only serves as an educational tool but also saves a lot of time and money that are spent on debugging by smart card manufacturing companies. In addition to that, technological advances in the smart card world can also be sped up with the help of this device for research and testing.

The hardware design of the project has been divided into 3 main sections, the snooping module, the communication module and the processing module. The snooping module taps the data, the communication module formats, encodes and transfers the data to the processing module, and lastly the processing module translates the data into useful information and display's it.

The project has been divided into two major milestones where the first one was to set up the snooping device prototype with the processing module being a Computer. The second milestone on the other hand is the final design itself which would be to replace the Computer with a PALM in order to make it portable and affordable.

This report gives a complete and detailed illustration on the hardware and software design process for the Universal "Chip Based Smart Card" Snooping Device. All aspects from the design decision, the underlying transmission protocols and also programming logics have been dissected and elaborately explained in the report. The design presented is a complete working device that fulfills all the objectives that have been set. The hardware device is certainly good enough to be marketed as it works flawlessly and achieves the most important objective of the project, which is to obtain the data transfer between the card and the smart card reader.

ACKNOWLEDGEMENT

The author's heartfelt gratitude is forwarded to:

- his internal supervisors, Mr. Patrick Sebastian and Mr. Zainal Arif Burhanudin, for their selfless imparting of knowledge and advice which guided the author throughout his final year to successfully achieve the requirements of his final year project;
- his lecturer, Mohd. Zuki Yusoff , for his creative ideas and input on the methods of improving the project.
- his external supervisor ,mentor and friend, Mr. Marc Talbot ,for his guidance and help that was given throughout the project duration.
- his lab technicians, Mr.Isnani, Ms.Siti Hawa. Mr.Farid, and last but not least Mr. Zairi for their kindness ,patience and willingness to lend a helping hand in order to help obtain devices or to operate the machinery in the lab
- his parents, Mr. and Mrs. Denamany, brothers, Darshan and Rubiin for their unconditional love and support which constantly propelled the author to strive for excellence;
- his friend and advisor , Jim Rees who is willing to respond and help him in time of need by shedding light on matters that are were troubling and hard to comprehend;
- and all the others whose names the author has failed to mentioned on this page, but has in one way or another contributed to the accomplishment of this project.

CHAPTER 1

INTRODUCTION

1.1 BACKGROUND OF STUDY

1.2 PROBLEM STATEMENT

1.3 OBJECTIVES

1.4 SCOPE OF STUDY

CHAPTER 1
INTRODUCTION

1.1 BACKGROUND OF STUDY

In today's modern world, smart cards play a major and significant role in making daily activities easier and more convenient. Regardless whether it helps in the communication sector, banking sector or in terms of security, its usefulness is undeniable hence making its demand grow as we head towards the future.

When an end user purchases a smart card from a smart card company, it *normally*[1] comes hand in hand with smart card readers. A company that has specific application that they would like to implement in the cards usually does the purchase of smart cards and smart card readers in bulk. Hence, it is a norm for the smart cards providers to provide smart card technical support for the purchasing party in order to help integrate the implementation of smart cards in to the systems that the customers have designed.

From an educational point of view, there are many individuals these days regardless whether from the industries or institutes of higher learning, that are trying to push their way into the world of smart cards, as it is part of today's cutting edge technology. Learning the method of manipulating smart cards could boost an individual's market price, as it is a tough area to comprehend and also because there are not many people in the world today who are well versed with this technology.

[1] Some smart cards have the reader embedded in the chip. Most cards that are given with readers are Access cards that do not have embedded readers.

1.2 PROBLEM STATEMENT

In reality, the process of providing smart card technical support for the integration of the customers system with smart cards will definitely have all kinds of smart card and smart cards reader related problems due to various common errors and occasional unforeseen errors. These problems are a norm during the implementation of a new system that features smart card technology. Considering this fact, there is a need in **many** situations where it would make the troubleshooting process easier if one would be able to monitor the data transfer between a smart card and a smart card reader for debugging purposes. The data transfer between the smart card and the smart card reader is basically the lowest [2]comprehensible programming level that can be accessed, as it is the transfer of APDU's[3].

This is where the problem arises, as a gadget, which features these capabilities are extremely expensive[4], if not unavailable. SchlumbergerSema for instance, is a smart card company that markets many innovative smart cards and smart card readers that are used worldwide today. And just like most of the other smart cards companies, they do not own or have a gadget that performs the above stated functions, as the technology is non-existent. Smart card developers and technical support engineers would have certainly save not only time, but also millions of dollars if such a device were to be engineered.

Besides that, this gadget can also function as an analysis tool for "non-computer" smart card readers. Hence, if any equipment were to have a smart card reader, this gadget could be used for troubleshooting purposes. All in all, it would function as a universal analysis tool for any device with a smart card reader from PC's, ATM Readers to customized smart card operated systems. In fact, smart cards technological advances that are normally tedious to test and debug could be made very much easier with this tool.

In addition to that, such a device could also make the process of understanding smart cards and its programming methods easier and much more effective as one would really be able to see the communication between the card and the reader. The steep learning curve could be instantly simplified with the existence of a tool like this.

[2] Refer to Appendix A
[3] Application Protocol Data Unit
[4] Approximately RM 40000 (comes with a PC attached to it)

1.3 OBJECTIVES

The following are the objectives of the **Universal "Chip Based Smart Card " Snooping Device** project:

- **To engineer a Smart Card / Smart Card reader analysis tool that is able to help boost technological advances in the smart card world.**

  The Snooping Device will most definitely help engineers and developers speedup their progress in terms of creating new hardware or software that is smart card related. The pleasure of being able to actually view the data transfer at the lowest level would certainly help both in the development and the testing phase.

- **To make the process of debugging the smart card readers and smart cards (simultaneously) at the lowest programming level possible, if not easier.**

  The debugging process when it comes to smart card reader and smart card are extremely tedious as there is nothing that is visible to the users eye except what is displayed on the monitor. Smart card manufacturings companies spend millions of dollars and a lot of time in order to detect and solve these bugs. The snooping device provides an opportunity for error detection to be made ridiculously easy as a study on the log of the data transfer basically pin points the exact problem that is encountered.

- **To enable developers and permitted parties to view and understand the communication between the smart cards and the smart card readers easily.**

  Smart card communication and the protocols used in order to make the communication possible is an area that takes much time and patience to comprehend. For those who are attempting to understand these topics, the snooping device serves as a very practical and effective short cut. As it is able to display the exact bytes that are transferred, it would make the process of understanding the format of the data and the relationship between the commands called and the bytes transferred much easier. Hence, it most definitely serves as an educational tool.

1.4 SCOPE OF STUDY

This project basically involves creating an electrical gadget that is able to understand and display the commands that are sent to and from a smart card or a smart card reader. The project can logically be divided into two sections, the software portion and the hardware portion. For the first half of the project duration, the objectives have been set to develop a snooping device that is linked to a Personal Computer (PC). Work on the second half of the duration would involve creating an enhanced portable snooping device that uses a PALM pilot as the processing and display module.

Based on the two targets that have been set, the software portion involves programming with Visual C++ for the first half of the project duration where else a detailed level of PALM programming for the final design. The hardware portion on the other hand involves RS232 serial cables and a significant amount of hardware design in order to create a proper snooping module for communication. As for the processing mechanism, the first prototype would require a PC where else the final design would need a PALM Pilot (PALM IIIxe). Smart card readers and smart cards would also be items that will be frequently used throughout the project execution. Other additional gadgets like level converters (MAX232) are default gadget that must be used in order to properly format the data that is transmitted. The MAX232 chip is used in both the hardware architectures and its function in both designs are identical.

In short, this project involves both an equal mix of software and hardware. The first prototype that was created was done in order to act as a stepping stone towards achieving the second and final hardware model. Please refer to Appendix B for the project Gantt chart that was created as a guideline for the first half and second half of the project duration.

CHAPTER 2

LITERATURE REVIEW AND THEORY

2.1 LITERATURE REVIEW

2.1.1 Smart Card in today world

2.1.1.1 In-built security

2.1.1.2 Pioneering E-Business Growth

2.1.1.3 Making multi application cards a reality

2.1.1.4 Managing Multi Applications

2.1.1.5 A Secure Future

2.1.2 A PALM Introduction

2.1.2.1 Introduction

2.1.2.2 What is a PALM Organizer?

2.1.2.3 PALM Software

2.2 THEORETICAL REVIEW

2.2.1 Smart Card Communication

2.2.1.1 Smart Card Insertion and Activation

2.2.1.2 Data Transmission

2.2.2 Asynchronous Serial Transmission

2.2.3 RS232 Interface

CHAPTER 2
LITERATURE AND THEORETICAL REVIEW

2.1 LITERATURE REVIEW

2.1.1 Smart Card in Today's World

Today, the multi-function smart card is firmly established as the basis for a vast portfolio of e-business services and products. Even more significantly, the smart card is at the forefront of empowering a host of mobile services – m-banking, m-email, and a multitude of other e-applications. The smart card is pioneering tomorrow's technology today, and still offers the most powerful combination of security and multi-functionality that meets the needs of today's service providers.

Since the 1970's the smart card has grown to become one of the leading technologies underpinning a whole world of varied and complex transactions. As it grew to dominance in the banking and finance arenas, the arrival of the GSM mobile phone was the real breakthrough that brought the smart card global pre-eminence. The SIM card provided a highly secure, personal ID for subscribers whilst providing network operators with control to transfer data as necessary. The arrival of the SIM Toolkit took the development of the smart card further. With SIM Toolkit, the card holder could download application programs via their mobile phone, enabling quick access to information such as travel, weather and stock exchange reports. Today, subscribers can now send and receive e-mails, whilst on the move. As the smart card continues to evolve, the multi-application smart card brings the only realistic option for managing multiple electronic transactions. The market opportunities are vast; a host of service providers want to work with network operators to deliver value added services to customers via the Internet and mobile enabled e-business.

2.1.1.1 In-built security

The smart card revolution has been propelled by the innate security the technology provides. Multi-application smart cards have demonstrated they can deliver highly secure transactions and enforce true protection between applications held within the card itself.

Their powerful encryption and digital signature capabilities are ideal for the emerging new technology sectors. Smart cards have built-in tamper proof qualities, and PKI (public key infrastructure) incorporating digital signatures) functionality embedded in their chip - all of which are essential to creating a totally secure environment for transactions.

2.1.1.2 Pioneering E-Business Growth

Smart cards are a cost effective, secure way to manage transactions electronically. They have become pivotal in the exploding e-commerce and e-business revolution. Today, one billion smart cards are in use – and by 2001 analysts project there will be 3.4 billion cards world-wide. The chip-based card opens the way to a single card managing multiple applications, again critical to the delivery solutions sought by service providers. Today, the multi-application card is already demonstrating the overall flexibility it offers manufacturers, issuers and users alike. A multi-application card can automatically update new services and existing applications. It can change and store user profiles for each application – and be accepted by a range of devices, including PC, POS, mobile phones and PDAs. For the future, multi-application cards are set to become the cardholders' personal ID. The multi-application smart card will become the route for individual's to receive personalized information services, and gain access to a range of services including banking, e-cash, ticketing, and loyalty programs. National government and state required data may also be simultaneously held on the card – driving license, passport and national identity information, for example. The multi-application smart card enables user to access a plethora of applications, together with individualized biometrics, all integrated into a single card.

2.1.1.3 Making Multi-application cards a reality

The smart card industry has driven major initiatives to support multi-application cards. There are now agreed industry standards to support operators, service providers, integrators, content developers, banks and developers that ensure inter-operability, application and key download. Integrity of downloaded applications is ensured by a certification process, developed by Oberthur Card Systems and based on the Visa Open platform specification. This authenticates new applets, defines operating rules and the

security mechanisms between applet issuer and the card issuer. Dedicated e-business smart cards, based on Java technology and again developed by Oberthur, enable several applications to be active simultaneously – ideal for payment, e-commerce and interactive environments. Applications such as identification-authentication-signature, debit/credit and e-purse payment, as well as value added services for loyalty, ID and health can all now be run at the same time.

2.1.1.4 Managing Multiple Applications

Using open platform technology, Oberthur has created a Card Management System that enables issuers to offer customers a range of services via a single card, and to manage these remotely. Based on Oberthur's Visa Open technology, the system manages the entire life cycle of the card, from production, card profile modification, application and key downloading to application removal – all conducted within a maximum security environment. An integral management system is critical to the successful operation of a multi-application smart card. Based on a system of databases, it contains information on the card – technical data, association profiles of the products and customer. It contains an application database, classified by service provider, and covering information related to loading the application. Key management contains master keys in a secure hardware module, and data to support key generation. A billing module tracks all information and transferred data. Finally, an audit database records all connections to the database, together with information relating to the behaviour of users.

2.1.1.5 A Secure Future

The dramatic growth of e-commerce and m-commerce, coupled with the convergence of IT, telecommunications, service providers, consumers and network operators means there is a massive demand for a single card to deliver secure, managed transactions in an open, platform independent environment. E-business is changing the way we do business. The need for secure e-payment and e-commerce is being met by today's highly secure, multi-application smart card. Supported by an open standard, scalable architecture that ensures inter-operability and co-operative activities between all parties who work with card

issuers to deliver services via the card, the multi-application smart card is truly enabling today's electronic transactions.

2.1.2 A PALM Introduction

2.1.2.1 Introduction

Released in 1996 from an unlikely source -- US Robotics, a modem manufacturer -- Palm Organizers now enjoy the dominant position in the Personal Digital Assistant (PDA) market place with approximately 80% of the market. It's not been an easy climb for the Palm, however, having to overcome Apple's Newton, Microsoft's Windows CE, and several smaller players like Psion.

The key to the Palm's success has been its simplicity and open development options. A lot of thought went into the PalmOS and applications, to ensure that people could do things quickly, efficiently, and without unneeded eye-candy that added nothing to functionality. The fact that anyone could develop applications for the Palm meant that developers were attracted to the platform in droves, creating a huge inventory of commercial, shareware and free applications.

There are a wide range of Palms on the market today, each targeted for a slightly different audience. The III series is the workhorse variety; these handhelds tend to be the cheapest and thus the most popular. For example, the IIIe, with 2 megabytes of RAM, can be found for less than RM 600 now. The Palm V series is targeted more towards executives, being slightly smaller, with a metal case and built-in rechargeable batteries.

In addition to handhelds built directly by Palm Computing, there are also third-party manufactured units such as the Handspring Visor models or the IBM WordPad. They all are running the PalmOS, licensed from Palm, and have the same buttons, touch-screen and writing area. Each has different amounts of built-in RAM and expansion abilities, and some will have flash ROMs that will let you upgrade your OS.

2.1.2.2 What is a PALM organizer?

Palm Organizers are full computers, but tiny enough to be held in the hand and designed to be used to help people stay organized. Most models are approximately 3 by 4.5 inches and about 3/4 of an inch thick. They have touch-sensitive displays that are 160 pixels

square; depending on the model, these will be either plain black-and-white, grey-scale, or color. Instead of a keyboard, there's an area beneath the display where a special kind of handwriting, "Graffiti," is used. In addition to this, there are four "soft" buttons for "Home," "Menu," "Calc," and "Search." Lastly, there are physical buttons for "Calendar," "Phone List," "Lists," and "Memos," plus scrolling and power buttons.

Most Palms have processors which are about twice as powerful as the first model of Macintosh computers, although some newer models are even faster. Most models are powered by a pair of AAA batteries, while the higher-end versions have built-in rechargeable. Battery life can provide weeks of regular use. While not something you'd run an RC5 key search on, these devices are certainly powerful enough for most handheld applications.

It's important to realize that Palms are not intended to replace a desktop or laptop, with their full environments, but instead are designed to be satellite computing devices supporting people while they're away from their desk. All Palms have a serial port which is used to synchronize information between the Pilot and the desktop by way of an adapter cable or cradle. While on the road, a modem can be used instead to update information. Some Palms also have an infrared (IR) port, which can be used to communicate between the devices and desktop machines if they're appropriately configured. And of course, wireless models can always be connected, providing they're in a service area.

2.1.2.3 PALM software

A key feature of the Palm design is that new software can be uploaded to the devices, supplementing or completely replacing the pre-installed software. The devices come with date book, address book, to-do list, memo pad, e-mail, and expense applications built in, with each application reading and writing well-documented database files. Enhancing a Palm simply involves finding an application you want to run and uploading it. There are lots of applications available -- some commercial but also a great many that are free.

Palm devices don't have a hard drive, so everything is stored in a nonvolatile RAM drive. Palm applications are simply files in this file system, ending in .prc, and sit alongside any database files they create, usually ending in .pdb. The "Applications Launcher" presents

the user with a list of all the .prc files on the Palm, with applications optionally categorized to the user's preferences.

This means that managing software is quite easy, since each application usually involves uploading just one file to the Palm, plus one or more database files. Backing up the device involves copying and saving these same files. On the Internet, Palm software is often distributed either packed into an archive format, or simply as an uncompressed .prc. Installing new software can be as easy as downloading from the Web with a browser and then uploading to the device.

[Source: Chris Halsall, http://preilvnet.com]

2.2 THEORETICAL REVIEW

2.2.1 Smart Card Communications

ISO 7816: Part 3 defines the electrical signals and transmission protocols. It describes the relationship between the smart card and the reader as one of a master (reader) and a slave (smart card). The reader establishes communication by signaling the smart card through the electrical contacts on the card. The smart card responds accordingly. The communication channel is single-threaded and so once the reader has issued a command to the smart card, it is blocked until a response is received. Appendix I illustrates the communication between the smart card and the reader through a series of state transitions.

2.2.1.1 Smart Card Insertion and Activation

Power is not applied to any of the contacts when a card is inserted into the reader. The reason for this is that a card could be seriously damaged if power was applied to the wrong contact. This could easily happen if a card were inserted into powered contacts. Instead an edge detector is used in order for the reader to determine when a card is properly aligned with the contact points. When the reader detects that the card is properly inserted, it applies power to the card. The smart card is powered up according to a well-defined sequence as shown on the timing diagram in Figure 2.1

The contacts are first brought into an idle state. This is characterized as being when the power (Vcc) is set high to a stable operating voltage of 5v. (An initial power setting of 5v

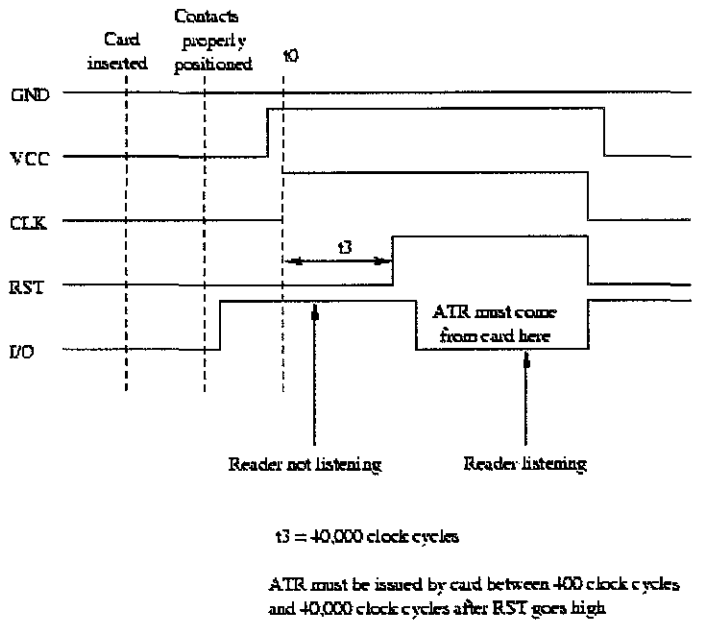


Figure 2.1: Timing Diagram of Smart Card Power up sequence

is always applied even though some microprocessor chips operate at 3v when in an I/O state.) The I/O contact is set to a reception mode on the reader side and a stable clock (CLK) is applied. The reset line should be in a low state and remain low for at least 40,000 CLK cycles before the reader can initiate a valid reset sequence. The reader then sends a reset signal to the card by setting the reset (RST) line into a high state. This signals the card to begin its initialization sequence.

Different cards may use varying specific initialization operations, however they should always result in the sending of an answer to reset (ATR) from the card to the reader. The time constraint on the first byte of the ATR being received by the reader is 40,000 clock cycles. If the ATR is not returned in the prescribed time, the reader begins a sequence to power down the card. In this sequence, the Vcc, RST, CLK and I/O lines are set low. Each successive byte of the ATR must be received by the reader at a minimal rate of 1 byte per second.

2.2.1.2 Data Transmission

The I/O line carries a single bit if data per unit of time defined by the CLK whose value depends on its voltage relative to GND. The convention of whether to use +5v for a bit

value of 1 or to use 0v is conveyed to the reader through the "initial character" of the ATR, also known as *TS*. The I/O line is always in the high state prior to the transmission of a character. It takes 10 bits to transfer 1 byte of data across the I/O line: the first bit is always a "start bit" (low state) and the last is a parity bit. The parity for each byte transferred should be even - the total number of bits in the byte whose value is 1 (incl. the parity bit) must be an even number. The following TS character indicates that the card uses an "inverse convention" i.e. H corresponds to a 0 and L corresponds to a 1: (H)LHHLLLLLLH. A TS character of the form (H)LHHLHHHLLH signals that the card uses the "direct convention" where H corresponds to a 1 and L to a 0.

The bit ordering in each byte is also controlled by the convention. In the inverse convention, the first bit following the start bit is the high-order bit of the byte. Whereas in the direct convention, the first bit following the start bit is the low-order bit of the byte. Successively higher order bits follow in sequence.

The communication channel to and from a smart card is *half-duplex* - *data* can either flow from the reader to the card or from the card to the reader, but not both at the same time. The significance of this is that the smart card and the reader must be synchronized. If both reader and card transmit at the same time then data will be lost. Moreover, if both are listening then the system will enter a deadlock situation. During the power-up sequence, both the reader and the card enter a receive state in which both are listening on the I/O line. Once the reset operation is completed the card enters a send state (to send the ATR to the reader). After this, both ends of the channel alternate between send and receive states.

The CLK and I/O lines are capable of supporting a wide range of data transmission speeds. The speed used is conveyed from the card to the reader via an optional character in the ATR. The transmission speed is set by establishing a "one bit time" on the I/O line, this means that an interval is established at which the I/O line can be sampled in order to read successive bits. This time is defined as an *elementary time unit* (etu). The etu during the ATR sequence is always defined to be:

etu = 372/CLK frequency

The CLK frequency is always between 1Mhz and 5Mhz - the frequency selected is generally such that the initial data transfer rate is *9,600 bits per second (bps)*. A typical smart card chip is capable of transmitting and receiving data at speeds up to 115,200bps. However, the data channel can be noisy and reliable communication is more important than high-speed communication.

2.2.2 Asynchronous Serial Transmission

Figure 2.2 shows the waveform corresponding to a single seven-bit character. In an asynchronous serial transmission system the clocks at the transmitter and receiver responsible for dividing the data stream into bits are not synchronized. The output from the transmitter sits at a mark state whenever data is not being transmitted and the line is idle. The term *mark* belongs to the early days of data transmission and is represented by a -12V in many systems operating over short distances.

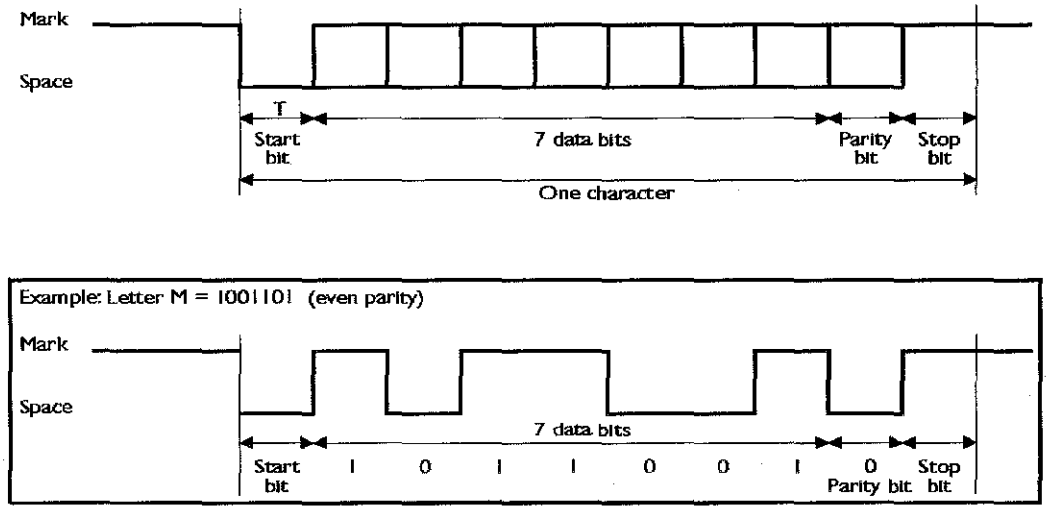


Figure 2.2 : Asynchronous Serial Transmissions

In what follows, a bit period is the shortest time for which the line may be in a logical 1 (mark) or a logical 0 (space) state. When the transmitter wishes to transmit a word, it places the line in a 0 state for one bit period. A space is represented by +12V. When the receiver sees this logical 0, called a *start bit*, it knows that a character is about to follow.

The incoming data stream can then be divided into seven bit periods and the data sampled at the center of each bit. The receiver's clock is not synchronized with the transmitter's clock and the bits are not sampled exactly in the center.

After seven data bits have been sent, a *parity bit* is transmitted to give a measure of error protection. If the receiver finds that the received parity does not match the calculated parity, an error is flagged and the current character rejected. The parity bit is optional and need not be transmitted.

One or two *stop bits* at a logical 1 level follow the parity bit. The stop bit carries no information and serves only as a spacer between consecutive characters. After the stop bit has been transmitted, a new character may be sent at any time. Asynchronous serial data links are used largely to transmit data in character form.

In short, **Asynchronous transmission** uses start and stop bits to signify the beginning and end of a transmission. This means that an 8-bit ASCII character with a parity bit would actually be transmitted using 10 bits. This method of transmission is used when data is sent intermittently as appose to in a solid stream. In the figure 2.3 the start and stop bits are in bold. The start and stop bits must be of opposite polarity. This allows the receiver to recognize when the second packet of information is being sent.

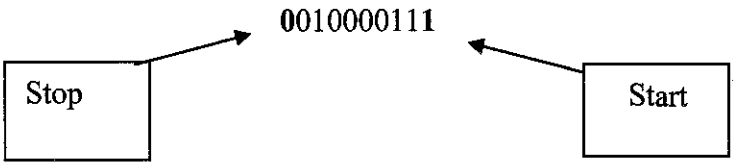


Figure 2.3: Illustration of start and stop bits

2.2.3 RS232 Interface

RS-232 has been around as a standard for decades as an electrical interface between Data Terminal Equipment (DTE) and Data Circuit-Terminating Equipment (DCE) such as modems or DSUs. It appears under different incarnations such as **RS-232C, RS-232D, V.24, V.28 or V.10** but essentially all these interfaces are interoperable. RS-232 is used for asynchronous data transfer as well as synchronous links such as SDLC, HDLC, Frame Relay and X.25

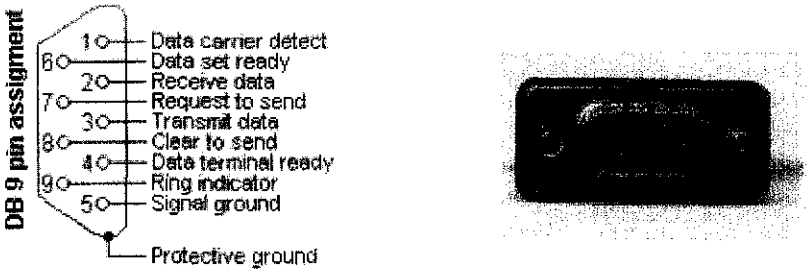


Figure 2.4: Standard Pin out for a DB-9 RS232 Connector

The essential feature of RS-232 is that the signals are carried as single voltages referred to a common earth on pin 7. Data is transmitted and received on pins 2 and 3 respectively. Data set ready (DSR) is an indication from the Dataset (i.e., the modem or DSU/CSU) that it is on. Similarly, DTR indicates to the Dataset that the DTE is on. Data Carrier Detect (DCD) indicates that carrier for the transmit data is on. Pins 4 and 5 carry the RTS and CTS signals. In most situations, RTS and CTS are constantly on throughout the communication session.

The clock signals are only used for synchronous communications. The modem or DSU extracts the clock from the data stream and provides a steady clock signal to the DTE. Note that the transmit and receive clock signals do not have to be the same, or even at the same baud rate.

The truth table for RS232 is:

Signal > +3v = 0

Signal < -3v = 1

The output signal level usually swings between +12v and -12v. The "dead area" between +3v and -3v is designed to absorb line noise. In the various RS-232-like definitions this dead area may vary. For instance, the definition for V.10 has a dead area from +0.3v to -0.3v. Many receivers designed for RS-232 are sensitive to differentials of 1 volt or less. The standards for RS-232 and similar interfaces usually restrict RS-232 to 20kbps or less and line lengths of 15m (50 ft) or less. These restrictions are mostly throwbacks to the

days when 20kbps was considered a very high line speed, and cables were thick, with high capacitance.

However, in practice, RS-232 is far more robust than the traditional specified limits of 20kbps over a 15m line would imply. Most 56kbps DSUs are supplied with both V.35 and RS-232 ports because RS-232 is perfectly adequate at speeds up to 200kbps.

CHAPTER 3

METHODOLOGY

3.1 BRIEF INTRODUCTION TO PROJECT METHODOLOGY

3.2 FLOWCHART OF COMPLETED ACTIVITIES

3.2.1 Theoretical Research Work

3.2.1.1 Hardware Related Research

3.2.1.2 Software Related Research

3.2.2 Hardware Design, Testing and Finalization

3.2.2.1 Hardware Design

3.2.2.2 Hardware Testing

3.2.2.3 Component Selection

3.2.3 Software Design

3.2.4 Prototype Construction and Troubleshooting

3.3 MAIN COMPONENTS DESCRIPTION

3.3.1 Smart Card Contact/Pins

3.3.2 MAX232CPE

3.3.3 Device Interface: RS232 Connectors

3.3.4 Palm Hardware

3.3.5 Palm Hardware

3.3.5.1 Dummy Smart Card

3.3.5.1.1 PCB Tracing

3.3.5.1.2 Milling

3.3.5.2 Connector

3.3.5.2 Overall Configuration

CHAPTER 3
METHODOLOGY

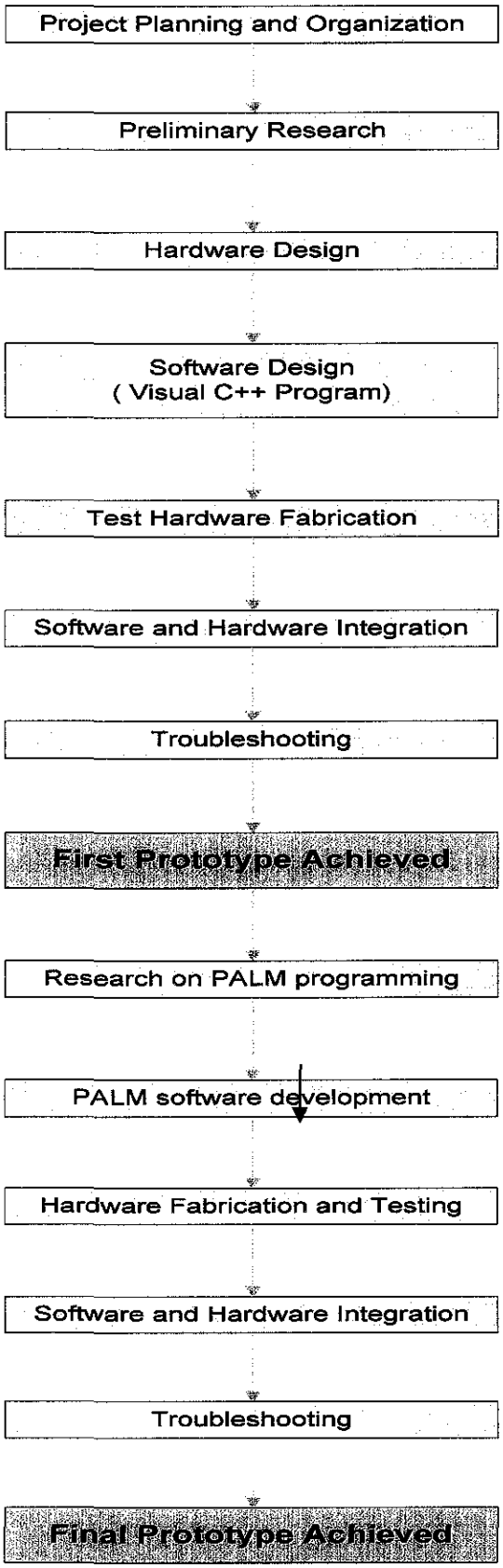
3.1 BRIEF INTRODUCTION TO PROJECT METHODOLOGY

The creation of the Universal "Chip Based Smart Card" Snooping Device has been divided into two major milestones in order to make the development process more effective and organized. The first target was to create a snooping device using a simplified architecture where the processing module for the design would be a personal computer (PC). The data will be processed in the PC and displayed using the monitor using a C++ program.

The second milestone on the other hand is the final design for the project where the processing module is converted into a device that is portable and convenient. This would logically require the use of a unique device, a PALM Pilot. The use of a PALM integrates the portability and affordability factors into the project's final design. The purpose of the creation of the first prototype is mainly to set the groundwork for the project research and to confirm that the project objectives (snooping of the data lines) are achievable using the proposed architectural design of the snooping module and the communication module.

By referring to the figures in Appendix C, the block diagram clearly illustrates the 3 main modules that have been identified in the hardware architecture. The first and final prototype only differs in the processing module where, as stated above, the PC is replaced with the PALM.

3.2 OVERALL PROJECT FLOWCHART



3.2.1 Theoretical Research Work

Theoretical research work is basically the theoretical reviews that have been done throughout the project duration in order to come up with the prototype and the final design. There were many books and website that have been referred to in order to obtain the desired information. As the project is divided into two portions, the hardware portion and the software portion, the research work done can also be categorized as such. Manuals and forums were also an important source of information as the information provided in these sources are relate to common problems that are normally encountered.

3.2.1.1 Hardware Related Research

The research done in order to come up with the hardware design mainly involved the identification of the use and functionality of various hardware devices that were looked into throughout the duration of the project. The research done on the hardware components was basically helpful in two ways. Firstly, it helped the process of learning about the different components available and its uses in order to be able to piece them together to create the prototypes. Data sheets and electrical websites over the web played a very important role in describing the components and its various uses. The second way in which it helped was to provide an understanding towards the smart card architecture and an idea of the possible action / processes that need to be done in order to obtain the desired output. For instance, the need for a level converter (MAX232 chip) would not have been apparent without the research that classified the requirements of the signal that is passed into the serial port. The research done was also a method of increasing the ideas that were created in the hardware design of the project while keeping it technically realistic and practical. The dilemma of whether to use a PIC micro-controller or a PALM for the portable processing module in the final design was solved by carrying out a structured and organized research on both devices.

3.2.1.2 Software Related Research

The software-based research done was wholly focused on establishing two types of programs that could accept serial data through a serial port, process the data and display it. The first program done was the Visual C++ program which was ran on the PC in order

to connect to the serial port and read the transmitted data. As for the final design, the software development portion involved writing a complete PALM application using specific softwares that compiled in special PALM development environments. The research carried out involved a thorough study of the various new softwares that were used and the specific methods of developing applications (especially for PALM). In addition, a lot of research was also in order to grasp an understanding of the serial port communication (PC and PALM) and the method as to which the program interprets the data transmitted through the serial cable.

3.2.2 Hardware Design, Testing and Finalization

3.2.2.1 Hardware Design

Throughout the project, there were several hardware designs that had to be engineered in as these devices perform unique functions that are specific to the project requirements. The following are some of the hardware designing that has been done:-

- Concept of tapping the signals from the actual smart card
  - At the preliminary stages of the project, many designs were made in order to create a hardware configuration that would enable the required lines from the smart card to be tapped and sent to the snooper whilst not interrupting or affecting the communication between the card and the actual reader. The design that was selected after careful consideration is the design illustrated in Appendix C, where the hardware was divided into three modules. At the point of design, the exact content of each module was not determined yet but the snooping method applied in the designed seemed logical and applicable.
- Dummy Smart Card
  - The dummy smart card is a totally new gadget that requires a unique design in order to fulfill its purpose. A complete illustration on the creation of the dummy smart card can be obtained from *section 3.3.5.1*
- The connector module
  - The connector in the snooping module also requires design work in order to implement a gadget that is able to ensure excellent connection

between the wires that are channeled out of it and the relevant contact on the actual smart card. The design here should be flawless as any problem with the data flow at this point will cause the whole system to fail.

- Communication module
  - This module is the module that is situated between the snooping and the processing modules. The design here involves putting together the required components that have been identified for it not to only function properly but also to properly connect to both its adjacent modules.

3.2.2.2 Hardware Testing

Hardware testing is an important part of the creation of the prototype due to two main reasons. First of all, by creating tests for a certain components, one would be able to fully understand and appreciate the functionality of the component. For the MAX232 chip for instance, Appendix D illustrates the circuit and the test method that was used in order to test the chip and learn the function of the chip. Practical work normally provides a more comprehensive understanding towards hardware components when compared to basic theoretical reading.

The second and more important purpose of hardware testing is to verify certain assumptions that were taken. In any design process, there are many assumptions that have to be taken in order to move forward with the project. These assumptions however can be verified by performing specially designed tests.

For instance, prior to starting the research on the required hardware components and the software portion of the project, a very general hardware design was required as the basis of the research. In other words, the method as to how the data was going to be tapped by the snooping device needed to be outlined. After considering many alternatives, the most outstanding design was selected (please refer to Appendix C) with certain assumptions made. The design specified had only one main property that had to be verified before it was finalized as the final design. A test was carried out in order to verify whether or not

the tapping device from the smart card is a logical and possible method. As illustrated, the design involved tapping the I/O pin of the smart card in order to get the data that is transmitted to and from the smart card. However, whether or not this was possible was yet to be verified. Considering the fact that there were no aluminum foil available at that point of time, a smart card reader was dismantled and the I/O pin of the smart card reader was tapped using wires. This is the same principle as tapping the I/O pin on the smart card as both these surfaces are in direct contact. This signal was then sent to a digital oscilloscope where the data transfer was studied. From the results of this test, the design was then verified and research on the rest of the components was started.

3.2.2.3 Component Selection

During the hardware design process, based on research, decisions have to be made regarding the components that are to be used. The decisions made are crucial as a wrong one may cause the project development to take a turn into the wrong direction. Factors that have been taken into consideration during the selection are issues such as cost, availability, practicability and most importantly workability. During the project execution, most of the components selected have been used. Many changes in hardware specification have been made during the transition of the process from the first prototype to the final product. These changes will be elaborated on in the discussion section.

3.2.3 Software Design

The software design involved creating a Visual C++ (1st prototype) and a PALM program (final design) that functions to receive the data from the serial port and display it in a meaningful manner. Besides reading up on documentations that are available on the web, the most effective manner used in order to create software was to actually do the coding and incrementally build the program up step by step. Examples were obtained from online program banks and studied.

Generally an evolutionary software development approach was adopted in order to create the programs. Development of the programs were broke down into portions in order to simplify the coding before putting them together. For the Visual C++ program, there

were three portions to it where the first objective was to just develop a program that is able to receive serial data. The second goal was to create a subroutine in order to convert the data into meaningful information. The final stage was to further enhance the program in order to display the information in a more meaningful manner.

For the PALM application on the other hand, 3 separate programs were created where the first one implemented dynamic fields. The second program on the other hand incorporated the serial library functions in order to create a program that is able to receive data via the PALM serial port. The third program on the other hand implemented the method of using and controlling scroll bars. Lastly, a final program, the snooping program was developed which incorporated all the three sample programs that were previously developed. It should be noted that the development of the PALM programs took more effort as the there were many new PALM development softwares that had to be mastered prior to actually developing an application. A complete explanation on the softwares used and its functions can be obtained from Appendix E.

3.2.4 Hardware Fabrication and Troubleshooting

One of the main activities of throughout the duration of the project was the work done in order to construct the specified hardware design. The first design was constructed in the middle of the first half of the project for the purpose of troubleshooting the Visual C++ software that is being created and also to verifying the design specifications based on the research. Many changes have been made since in order to overcome various problems and perfect the circuit so that it would be more reliable. The fabrication of the hardware were done involved various activities such a PCB fabrication, milling, soldering, circuit analysis and etc. Since most components in the design are not available commercially, they were all fabricated in the lab. In order to troubleshoot the hardware, various tests methods were designed in order to ensure that there were no flaws in the design. The test methods implemented were made to be as simple as possible in order to make the error detection simpler.

3.3 MAIN COMPONENTS DESCRIPTION

3.3.1 Smart Card Contacts/Pins

The layout of the pins in all chip-based smart cards has been standardized based on the ISO7816-2 standard. The following is a simple diagram illustrating the contact pins on a smart card.

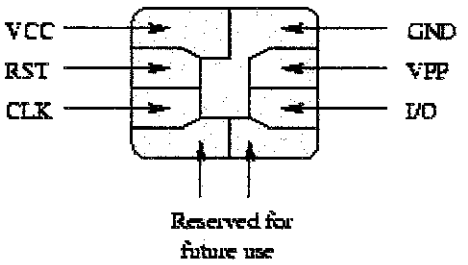


Figure 3.1: Smart Card Pin Layout

The smart card chip, or integrated circuit card (ICC) has 8 electrical contacts. They are referred to as C1 through C8. However, as shown above not all 8 contacts are electrically connected to the embedded microprocessor chip and therefore unused at the present time. The following table contains the contact definition according to ISO7816-2 standard.

Contact	Designation	Use
C1	**Vcc**	Power connection through which operating power is supplied to the microprocessor chip in the card
C2	**RST**	Reset line through which the IFD can signal to the smart card's microprocessor chip to initiate its reset sequence of instructions
C3	**CLK**	Clock signal line t hrough which a clock signal can be provided to the microprocessor chip. This line controls the operation speed and provides a common framework for data communication between the IFD and the ICC
C4	**RFU**	Reserved for future use
C5	**GND**	Ground line providing common electrical ground between the IFD and the ICC
C6	**Vpp**	Programming power connection used to program EEPROM of first generation ICCs.
C7	**I/O**	Input/output line that provides a half-duplex communication channel between the reader and the smart card
C8	**RFU**	Reserved for future use

Table 3.1: Smart Card Pin Functions

3.3.2 MAX232CPE

Given is the physical pin layout of all MAX232 chips.

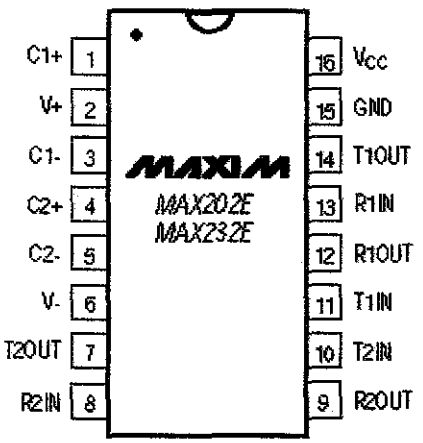


Figure 3.2: MAX232 Pin Layout

The purpose of the MAX232 chip is to convert the voltage levels and encode data. The serial port of a PC/PALM and the smart card both has different ways of representing the same data. The smart card represents its bits in voltage levels of 5V (high) and 0V (low). The serial port on the other hand represents a high as a voltage between –3V and –12V and a low as a voltage between +3V and +12V. Basically when changing the smart card data signal to the serial port data signal, the MAX232 magnifies the voltage to its appropriate level. The magnified signal is then encoded using the Non Return To Zero Level (NRTZ-L) encoding scheme. The reverse happen when the data is sent from the serial port to any other standard devices. Common names for MAX232 chips are *level converters*.

3.3.3 Device Interface: RS232 Connector

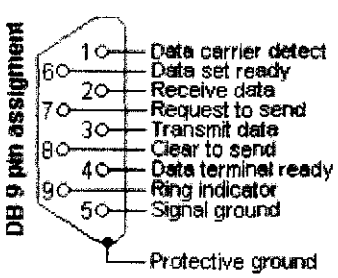


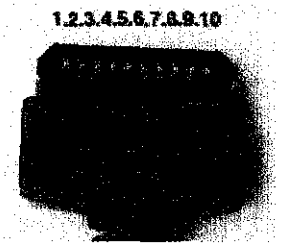
Figure 3.3: RS232 Serial Port Pins

The RS232 connector functions as the data transmission connector between the MAX232 chips and the serial port of the PC/PALM. As illustrated in the diagram above, there are nine pins available on the connector. In the design specification, only seven out of the nine pins available are used. The transmit pin and the ring indicator pin is not used.

3.3.4 PALM Hardware

The PALM that was used in the final design was the PALM IIIxe. This PALM is used to connect to the user's personal computer via a device called a cradle. The PALM is placed on the cradle and a wire from it is then connected to the serial port of the PC. Besides the cradle, a Palm HotSync cable could also be used. The cable is basically identical to the cradle except for the fact that instead of placing the PALM on the device, the connector is directly connected to the Palm's port (please refer to Appendix F for a graphical illustration)

Basically the port on the PALM that connects to the serial cable of the cradle or the Hot Sync Cable is actually a serial port. The only difference is that contacts on the PALM are all flat.



Pin No	Func.	Pin No	Func
1	DTR	6	CTS
2	Vcc	7	GPI1
3	RX	8	GPI2
4	RTS	9	
5	TX	10	GND

Figure 3.4: Pins on the HotSync Cable

Figure 3.4 clearly shows the function of each pin on the HotSynch cable. Basically, the port for the Palm device[5] also has those contacts. These contacts are then just wired to the female socket of a basic RS-232 connector so that it can be fitted into the male serial port on the PC.

[5] Refer to appendix F for the images

3.3.5 Dummy Smart Card and Connector Fabrication

The dummy smart card and the connector were basically two gadgets that had to be specifically engineered as they are not available in any store. These two gadgets were designed using various tools in the mechanical and electrical labs.

3.3.5.1 Dummy Smart Card

The dummy smart card is the component of the design that will be inserted into the target smart card reader. The dummy card would be connected to the "connector" (explained next) which would be connected to (or touching) the contact of the actual smart card. Hence, the dummy smart card is required to have all eight contacts fit exactly into the smart card reader with absolute precision. In addition to that, these contacts must also be connected to pins at the edge of the card in order enable the tapping of the signal lines[6]. This card is a very unique device that was created using PCB fabrication. It is actually a PCB board which has been modified to become a smart card. The following are the steps that have been taken in order to fabricate the dummy smart card: -

- Creation of the Gerber file using the ARES LITE[7] software
- Creation of the dummy smart card on a PCB using mechanical etching.
- Alteration of the width and height of the card in order to align the dummy smart card contacts with the smart card reader contacts.
- The thickness of the PCB was carefully reduced to the thickness of a smart card using a Milling machine.
- Holes were drilled into the smart card in order to place connectors (pins) on it.
- The PCB was cleaned using sandpaper in order to create a clean surface.
- The connector pins were soldered onto the clean surface of the PCB.
- The surface of the PCB was then coated with a thin layer of coating for protection.

The steps given were accomplished with the guidance of technicians from each of the respective labs. Snap shots of the dummy smart card can be seen in Appendix G. With

[6] refer to Appendix G for a picture of the dummy smart card
[7] gerber file included in attached CD

reference to the given fabrication steps, the following is a detailed description of some of the challenging processes.

3.3.5.1.1 PCB Tracing

The PCB tracing that was done required the implementation of special techniques as the desired layout of the PCB is not exactly a circuit. The only type of PCB tracing that was available in the lab at was mechanical etching. This meant that the PCB board was placed in a machine that would slowly strip off the copper surface of the PCB in order to separate the contact. The movement of the machine is based on the Gerber file that it loaded into it. A Gerber file can be prepared using several softwares..

The ARES Lite software was selected to design the tracing patterns as it was easy to use and quite sufficient for the creation of the dummy card as the software did not require circuit analysis. The tracing process with the software was extremely important as the measurement of the contacts that was provided had to be exact. Based on the ISO7816 standard, the 8 contacts on a smart card have default locations and all the smart card readers (which are normally based on the ISO7816 standard) are designed to comply with these measurements.

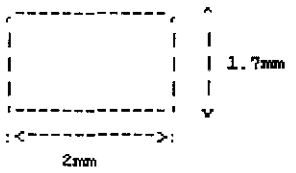


Figure 3.5: Minimal Contact Size

	A	B	C	D
C1	10.25	12.25	19.23	20.93
C2	10.25	12.25	21.77	23.47
C3	10.25	12.25	24.31	26.01
C4	10.25	12.25	26.85	28.55
C5	17.87	19.87	19.23	20.93
C6	17.87	19.87	21.77	23.47
C7	17.87	19.87	24.31	26.01
C8	17.87	19.87	28.85	28.55

ISO7816 location

Table 3.2: Measurement of Pin's Position based on Figure 3.6

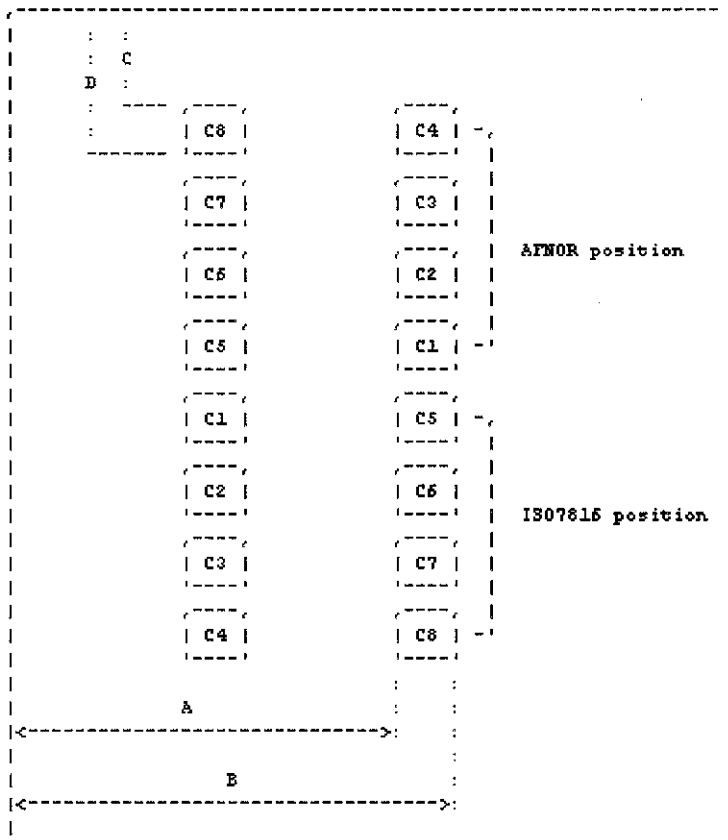


Figure 3.6: Pin's Position

After carefully creating the Gerber file based on the measurements above, it was downloaded into the PCB tracing machine and the PCB board was prepared. Upon completion, the PCB was then removed from the machine and the measurement process was done once again as the PCB would have to be cut to the proper width and height.

3.3.5.1.2 Milling

Once the width and height of the PCB board have been made to be identical to that of a smart card, the only remaining physical alteration is to customize the thickness of the PCB. This is important as most smart card readers have a slot which is not more than 0.1mm wider than the actual smart card thickness. A smart card generally has a thickness of 0.8mm. This is almost half the size of a normal PCB board. Hence, in order to slim down the PCB, the milling machine seemed to be the best tool that was available in the mechanical workshop.

The process of milling the card to become thinner was an extremely delicate job as the risk of the card breaking was very high given the consideration that the milling machine was not built to work on materials such as PCB boards. Another important consideration that had to be taken into account is the effect of warping. Warping is a phenomenon where the material in question (which is normally not very thick) begins to bend due to excessive force. With all the possible risk to the dummy smart card, many trial runs were attempted on unused PCB board before work was actually done on the dummy smart card.

In order to protect the card from the various destructive effects of the milling machine, the milling process was done step by step by working on a small area at a time. Only the area that was going to be milled would be exposed. The rest of the card would be mounted down using rubber stoppers in order to ensure that the card was always flat. Each portion of the card is pressed down unless milling was done on its surface. After carefully milling the card, the dummy smart card seemed to have emerged slightly thinner than expected and with a little bit a warping on its edges. Nevertheless, as far as its contacts with the smart card reader were concerned, all eight contacts were perfectly aligned with the dummy smart card and the size of the card was within an acceptable limit.

3.3.5.2 Connector

The connector is the device that helps establish a connection with the actual smart card contacts. It is a device that is supposed to function to tap all the contacts of the actual smart card and transmit them to the dummy smart card. Hence, in order to design such a device, the easiest method was to modify an existing smart card reader. The Schlumberger e-Gate connector was used as the target reader. The e-Gate was forced apart and its processing board was cut off. Wires were then carefully soldered to all the contacts that are used by the smart card reader[8]. These wires were then routed out of the reader and into a PCB board that is connected to the dummy smart card. Each wire was carefully matched with its corresponding contact on the dummy smart card. Finally, the e-Gate was glued back together using super glue.

[8] The smart care reader only uses six contacts. Two of the contacts are not used

This means that there is a direct connection between each contact in the dummy smart card to each contact in the actual smart card hence making the actual smart card reader unaware of the difference in the hardware configuration. Please refer to Appendix G for snap shots of the connector.

3.3.5.3 Overall Configuration

By connecting the dummy smart card to the connector, the snooping device is complete. For testing purposes, the device has been used on its own with out any snooping in order to examine its effectiveness. When a smart card is connected to a Reflex 20 smart card reader via the snooping module, the card seems to work extremely well and the reader does not indicate any problems at all as all operations verify that the reader is unaware and unaffected by the presence on the snooping device. This test was repeated with multiple reader and they all worked fine.

CHAPTER 4

RESULTS AND DISCUSSION

4.1 SYSTEM DESCRIPTION & FUNCTIONALITY

4.1.1 Snooping Module

4.1.2 Communication Module

4.1.3 Processing Module

4.2 SYSTEM DESIGN DETAILS

4.2.1 Smart Card Communication Details

4.2.2 Interfacing the MAX232CPE and the Serial Connector

4.2.2.1 MAX232CPE

4.2.2.2 Serial Port Connector (DB-9)

4.2.3 PALM Snooping Program

4.2.3.1 PALM Program Development

4.2.3.2 Program Overview

4.2.3.3 Receiving the Bytes

4.2.3.4 Byte Conversion Subroutine

4.2.3.5 Data Interpretation

4.2.3.4.1 ATR Standard Format

4.2.3.4.2 APDU Standard Format

4.3 ELABORATION ON THE FINAL OUPUT

CHAPTER 4
PROJECT IMPLEMENTATION, RESULTS AND DISCUSSION

4.1 SYSTEM DESCRIPTION AND FUNCTIONALITY

From the block diagram and circuits illustrated in Appendix B, it is clear that the snooping device can be divided into three modules, the Snooping module, Communication module and the Processing module. Each module has its own overall functionality as it consist of several components that have been interfaced together to achieve certain goals.

4.1.1 Snooping Module

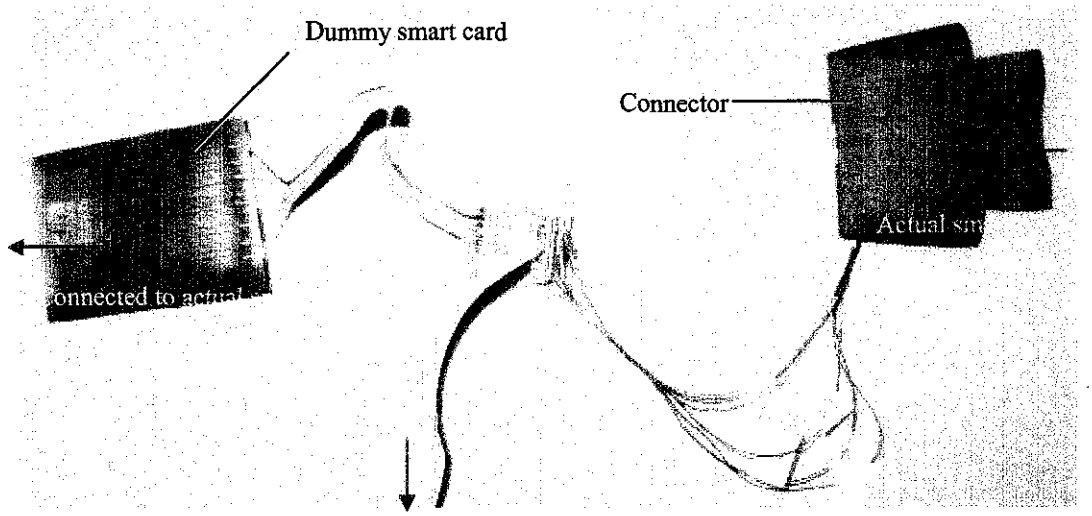


Figure 4.1: Snooping Module (Dummy Smart Card and Connector)

The snooping module basically consists of the:

1. Dummy Smart Card
   - ❖ The dummy smart card is actually a specially fabricated PCB board that has is the exact same size as an actual smart card and has eight contacts at the exact same position as the actual smart card. These eight contacts are all linked to the edge of the card where connector pins are connected to it. The connector pins will be used in order to link the contacts of the dummy smart card and the contacts of the actual smart card.

- ❖ The purpose of the dummy smart card is to make the target reader "think" that there is a smart card in it even though the smart card is not actually physically in the reader; it is just connected to the reader via the dummy smart card.

2. Connector
   - ❖ The connector is the interface that provides the link between the connector pins on the dummy smart card, and the pin contacts of the actual smart card. The main purpose of the connector is actually to make the design more robust and to make the snooping of various smart cards convenient. The connector has a slot (like a smart card reader) where the actual smart card is inserted. The pins of the smart card would then come into contact with a copper contact plate that is physically linked to the 8 pins on the dummy smart card. This gadget saves the user the trouble of manually connecting the 8 pins from the actual smart card to the dummy smart card.
3. Actual smart card
   - ❖ This is the target card that the user decides to snoop on. The actual smart card will be inserted into the connector as discussed above.
4. Actual Smart Card Reader
   - ❖ The actual smart card reader is the device that will house the dummy smart card. This is the smart card reader that is supposed to be able to communicate with the actual smart card.

When the actual smart card is inserted in the connector and the dummy smart card is inserted into the target smart card reader, communication between the actual smart card and the smart card reader (via the dummy smart card and the connector) will begin. All the signals from the smart card will be transmitted to the smart card reader through the physical connection that has been set up via the dummy smart card and the connector. The signals from the reader to the actual card also are transmitted using the same manner. As far as the smart card reader is concern, the communication that takes place is just like the normal situation of the actual smart card being inserted into the reader.

With that portion set up, the snooping circuitry then comes into play where certain contacts from the smart card are tapped in parallel with the existing circuitry. A smart

card has 8 contacts where there are normally 6 contacts that are used[8]. In order to snoop on the data transfer, there are four main contacts that are tapped, the I/O contact, V_{cc} contact, Gnd contact and the RST (RESET) contact.

The I/O contact contains the data that is transferred from the card the reader and vice versa. This line basically contains all the information that needs to be displayed as all communication takes place via the I/O contact, as it is a half-duplex line. The information regarding the data transfer is further illustrated in *section 4.2.1*. The RST contact on the other hand is the reset pin for the smart card chip. These two lines are linked to the Communication module, which will be discussed in the next section.

4.1.2 Communication Module

The communication module is the portion of the circuit that is in charge of transferring and formatting data from the snooping module to the processing module. The module consists of two main components, the MAX232CPE chip and the RS232 Connector. The functionality of these devices and some of its technical specifications can be obtained from section *3.3.2* and *3.3.3*.

The exact connections made for the communication module can be seen in the circuit illustrated in Figure 4.3(refer to Appendix H for clearer diagram).

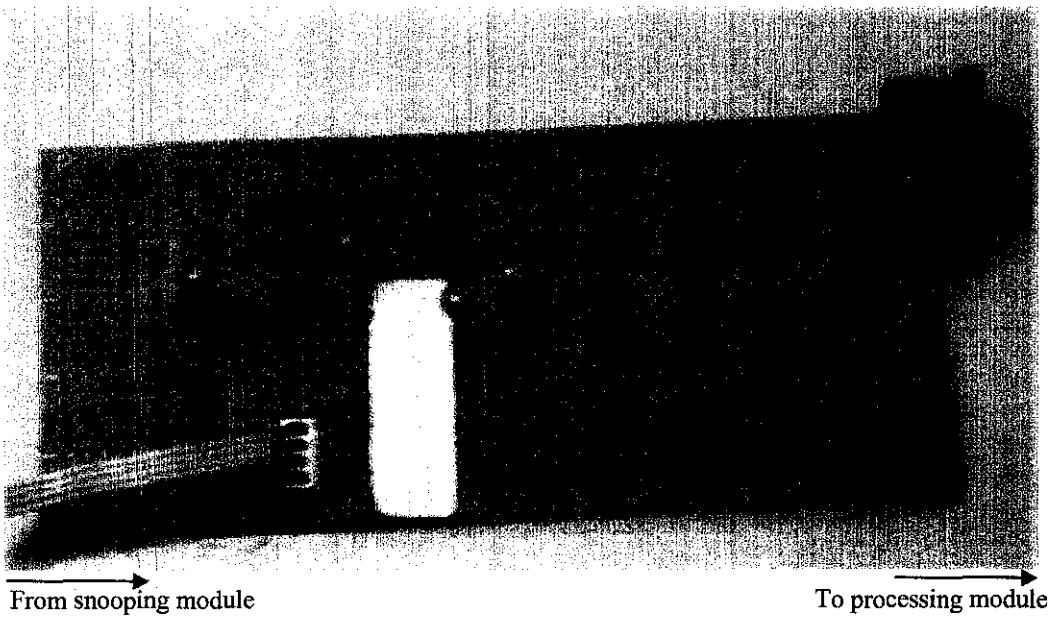


Figure 4.2: Communication Module

[8] refer to section 3.3.1 for details on the smart card contacts

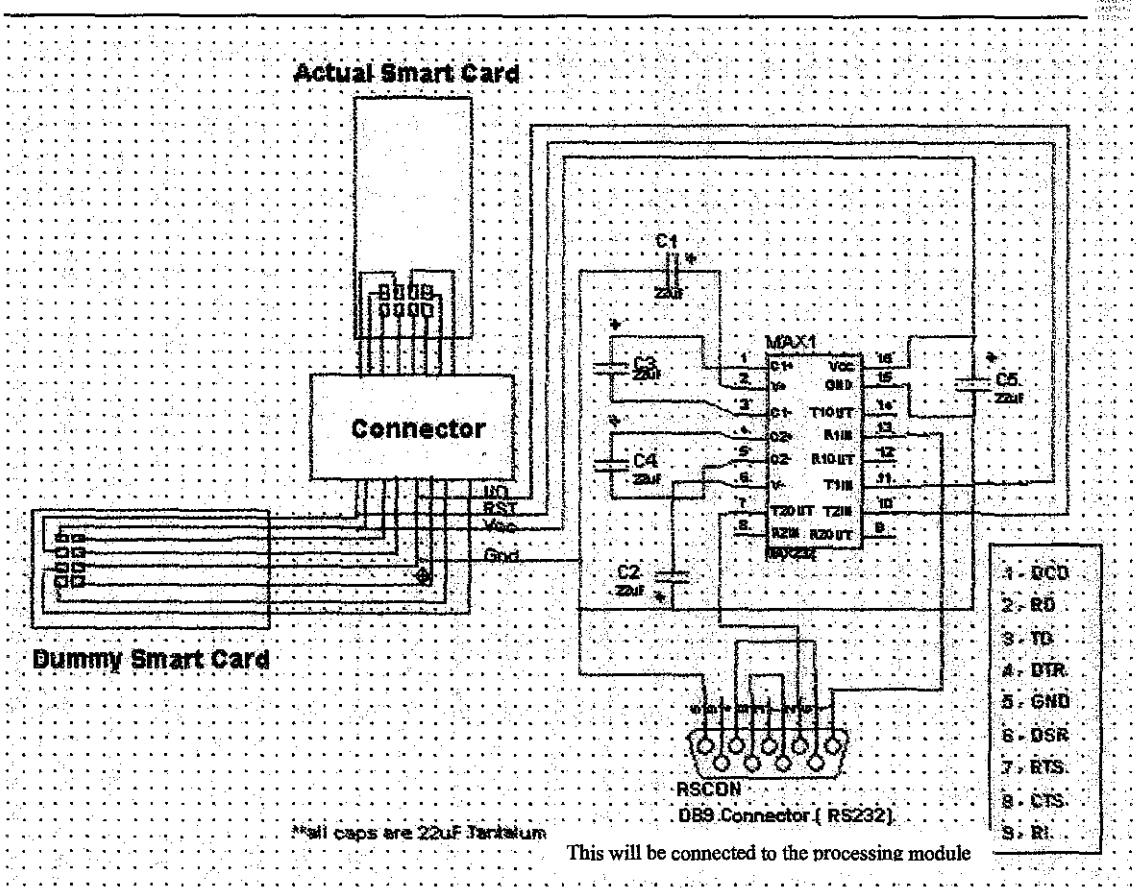


Figure 4.3: Schematics of the Snooping and Communication Module

There are four contacts that have been tapped from the card and brought into the communication module from the snooping module. The Gnd contact is connected to both the MAX232 and the RS232 connector (GND-Pin 5) for the sake of standardization. The Vcc contact is connected to the Vcc of the MAX232 chip. Hence, the MAX232 chip actually obtains its power source from the smart card 5V Vcc contact.

The I/O contact and the RST contact is connected to the MAX232 chip in order to convert the voltage level. The on both these contacts that are represented as 5V voltages (HIGH logic) and 0V voltages (LOW logic) are converted into a representation that is understood by the serial port of the processing module. The HIGH logic on the serial port is represented as a voltage between –3V and –12V where else a LOW logic is represented as a voltage within 3V to 12V. All data sent to the serial port must be converted into this format; hence this shows the importance of the MAX232 chip. Once the signals have

been converted, the I/O line is sent to the Receive (RX – Pin 2) Line of the RS232 connector.

The output of the RST line from the MAX232 on the other hand is sent to the Data Carrier Detect (DCD-Pin 1) Pin. The rest of the pins are configured as shown in Figure 4.1. A detailed explanation on the pin configuration can be found in *section 4.2.2.2.*

The RS232 Connector is then connected to the serial port of the processing module and data is then accepted by the module and processed.

4.1.3 Processing Module

The processing module server two main purposes; the first is to process the data that is sent in through the serial port to become meaningful information. The second function is to organize the information properly and display it in a meaningful manner. As mentioned, there are two types processing modules which have been used throughout the project. The first one is a PC where this was actually a prototype for the final design. In this module the serial port is controlled using a Visual C++ program that accepts the serial streams and translates the data into meaningful information. In the final design, the processing module that is used is a PALM device and the serial connector from the communication module is connected to the cradle or HotSynch cable of the Palm. In order to accept the serial data and process it, a Palm application has been developed using custom made softwares[9] that were designed specially for Palm Application Development. Both the softwares written for both the processing modules are almost identical in terms of its programs structure. The programming environment and the functions calls made in both programs might differ completely, however, the program implementation and the logics behind it is rather identical. In both programs, the initial stages of its execution involve the setup of the serial port of its respective hardware. Parameters such as the baud rate, start bit, stop bit and etc. have to be set to establish a successful communication. Then, the serial port is opened and the receive buffer is checked periodically[10] for incoming data. In the Palm application, the process of moving in a loop is a default characteristic that is implemented (refer to Appendix E for an explanation on the Palm

[9] please refer to Appendix E in order to get an overview of the Palm Application Development Softwares
[10] also known as polling

programming structure). Every time data is received, memory is allocated for the data in order to temporarily store it. The data is then manipulated in order to actually display the transmitted bytes.

In order to display a character, on the screen, the character must first be translated in to its ASCII[11] representation. ASCII is a code in which the numbers from 0 to 255 stand for letters, numbers, punctuation marks, and other characters. Hence in order to display the bytes that were received, they must first be converted into its respective ASCII representation. The ASCII representation for all possible incoming data is given in Appendix I.

The bytes that are transferred all comply with the standard format that is stated in the ISO7816-3 protocol. The first few bytes received upon connection is called the ATR, the Answer to Reset. The ATR has a predetermined format. The bytes that are received after the ATR until the point of disconnection are all called APDU's (Application Protocol Data Units). Further details on the byte to ASCII conversion process and the explanation on the received data is available the following sections. The detailed explanation on the programming of the processing module will be covered only **for the PALM device** as it is the final design. However, screen shots of the Visual C++ program can be obtained from Appendix J.

4.2 SYSTEM DESIGN DETAILS

4.2.1 Smart Card Communication Details

The smart card communicates with the outside world through the I/O contact. All communication with the smart card is classified as a half duplex communication where the data transfer can only take place at one direction at any point of time. The data is transferred to and from the I/O pin using asynchronous serial transmission[12]. The following are the asynchronous data transmission details of the card:

[11] American Standard Code for Information Interchange.
[12] refer to *section 2.2.2* for details an asynchronous serial transmission

Baud Rate	9600
Start Bit	1 Start Bit
Stop Bit	2
Parity Bit	Even parity

Table 4.1: Asynchronous data transmission details

With these settings made compulsory, all data transfer from and to the smart card must comply with these conditions. This indicates that the smart card chip actually has an in built Universal Synchronous Receiver /Transmitter[13] (UART).

There are four main contacts that are tapped from the smart card, the Vcc, Gnd, I/O and the RST contacts. The Vcc and ground contact are basically tapped in order to power up the MAX232CPE chip. The I/O and the RST contact are the lines that carry the important data that is needed. As stated, the purpose of the I/O contact is to enable half duplex data transfer to and from the card. Hence, when the snooping device taps the I/O line, it becomes possible to read all information transferred to and from the card with the condition that the signal is interpreted correctly. The I/O line is normally kept high when there is no data transfer (idle state) between the card and the smart card reader. The RST (reset) line on the other hand functions as a reset for the smart card. When the card is functioning, this line is always kept high. A reset can be applied to the card when there is a low to high transition in the RST line. A more detailed explanation on the I/O line and the RST line is illustrated in *Section 2.2.1.*

When a card is first inserted into the smart card reader, all lines will remain low until all the contacts are properly positioned. Once the contact are positioned properly the, card is powered up as the Vcc contact is brought high. The Vcc contact will remain high until the card is disconnected. This in turn means that the MAX232CPE chip, which is powered by the Vcc pin, will be activated and deactivated exactly when the card is. The smart card will then be RESET, as the RST line will be brought high after a certain period of time.

[13] UART functions to convert 8 bit bytes into the asynchronous data form by adding the start, stop and parity bits or vice versa.

The first action upon a card reset is that the card would then transfer a string of bytes known as the Answer to Reset (ATR) to the smart card reader The ATR sequence is responsible for initializing the physical communication channel between the reader and the smart card. It facilitates the definition and manipulation of a number of characteristics of the channel. In short, the ATR is actually a long number that identifies the smart card to the smart card reader and it also contains some control information that specifies the protocol and setting that the reader should adhere to in order to establish a successful communication line with the card.

ISO 7816-3 also specifies a more elaborate method of selecting a protocol known as the *Protocol Type Selection* (PTS) facility. The reader can negotiate with the card to obtain an optimum set of characteristics for the channel through the PTS. When the ATR has been transmitted, the reader would then evaluate, based on the protocol indicated in the ATR, whether or not a *Protocol Type Selection Request* (PTS) is necessary. If it is, the reader would execute the request and the card would reply with a *Protocol Type Selection Response*. Normally, optimum communication characteristics are usually through the ATR sequence without performing a PTS sequence.

After the completion of these sequences of byte transfer, the rest of the communication between the smart card and the smart card reader involves only the transfer of Application Protocol Data Units (APDU). APDU's are basically commands that are sent to the smart card in order to perform certain operations. In turn, the card would respond with to the command in order to indicate whether the message had failed or succeeded. The response may be prefixed by some data bytes if the command involved a request for certain information from the card. A summary of the whole communication discussed above is briefly summarized in Figure 4.4.

As illustrated in the figure, after the ATR and the PTS (if necessary) are sent, the reader (terminal) issues commands and the card will provide a response to each command. These actions are repeated over and over again until the card disconnects from the reader or it is reset. When the card is reset (RST line goes from low to high), the whole sequence repeats once again.

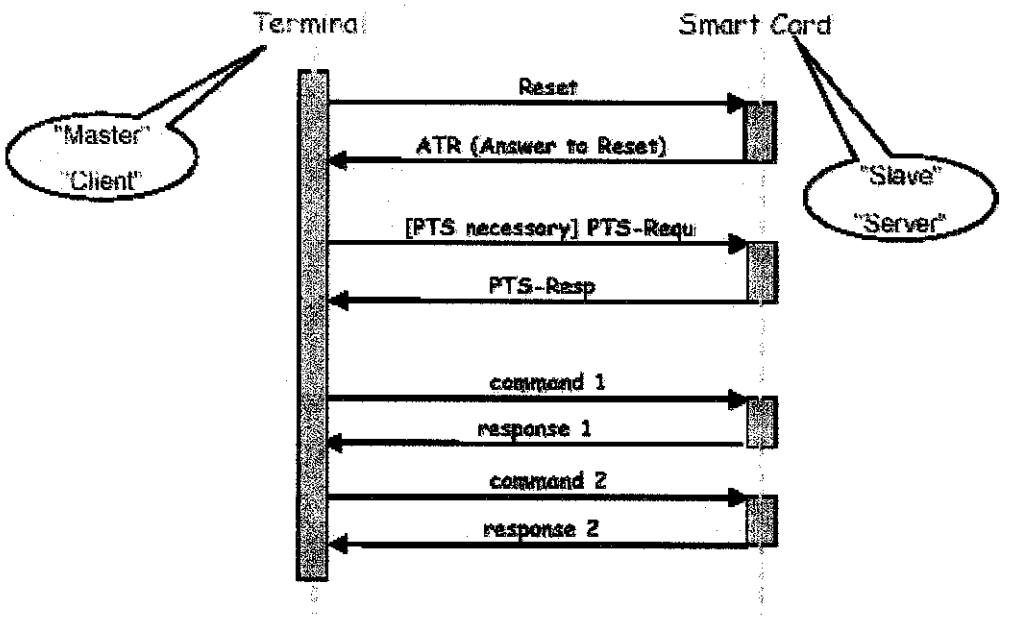


Figure 4.4: Smart Card Activation Sequence

4.2.2 Interfacing the MAX232 and the Serial port connector (DB-9)

4.2.2.1 MAX232CPE*

The MAX232CPE chip gets its power source from the Vcc and Gnd contact of the smart card. The chip functions to convert the high and low voltage levels from the smart card to its corresponding voltage levels according to the standard requirements of the serial port. There are basically two lines from the smart card that is transmitted into the MAX232CPE chip, the I/O line and the RST line. The I/O line is inputted through pin 10 where else the RST comes in through pin 11. The output for the I/O line and the RST line are on pins 7 and 14 respectively. The output for the I/O pin is the connected to the Receive pin of the DB9 connector. This pin functions to transmit the data into the serial port. Hence, all the data from the I/O contact is acctualy being transmitted into the processing module via the receive (RX) pin on the serial port. As for the RST line, the output signal from the MAX232CPE chip is connected to the Data Carrier Detect (DCD) pin. Explanations on the connections made to the DB-9 connector are given in the following section.

* refer to Appendix F for the schematics

The MAX232CPE circuit setup is a standard schematic from the data sheet. However, the only special element is the type of capacitors that are used. The standard specification recommends a 10uF electrolytic capacitor. However, for this project, the capacitors used were the 22uF tantalum capacitor. The reason these capacitors were selected was because tantalum capacitors are meant for high switching circuits and they are also much more reliable when compared to the electrolytic capacitors.

4.2.2.2 Serial Port Connector (DB-9)

The connector has 9 pins that are available. From these 9 pins, only 7 of the pins are used in order to establish the asynchronous data transfer required.

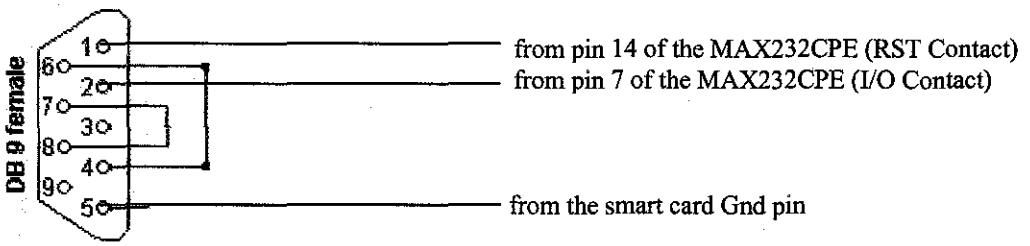


Figure 4.5: Connection to the DB-9 connector from the MAX232CPE chip

The following is a simple summary of the connections made and some brief explanation on the logic behind the connections.

DCD (Pin 1)

This pin is called the Data Carrier Detect pin. The purpose of this pin is to actually detect the presence or the absence of a carrier wave. For normal use o the DB-9 connector, if a carrier is present, it indicates that the transmission is line is "alive" (either receive or transmit) and that the port would have to be alert of possible data transfer. On the contrary, an absence of the carrier means that no data will be transmitted making the port is "dead". A simple analogy would be to assume this pin as the power supply for the serial port. Hence, by connecting the RST contact to this pin via the MAX232CPE, whenever the reset is low, the port is basically not powered and when it goes high the port is powered up and able to do the data transfer. Since the reset goes from low to high every time it is triggered, the serial port also is deactivated and reactivated.

RX (Pin 2)

The RX pin is the receive pin. This pin functions to receive synchronous or asynchronous data. The data from the I/O line is sent into this pin via the MAX2323CPE chip so that the processing module can receive the transmitted information.

DTR (Pin 4) and DSR (Pin 6)

The DTR, Data Terminal Ready Pin indicates that the DTE is ON and ready to communicate when it is asserted. The DSR pin on the other hand indicates that the DCE is connected and ready to transmit data when it is asserted. These two lines mainly function as handshaking lines in order to synchronize communication. Since handshaking is not used for the data transmission, these two pins have been connected together. This means that once the serial port is ready, it will assert the DTR line which in turn causes the serial port to think that the other party is also ready for communication as the DSR line is asserted simultaneously.

GND (Pin 5)

Pin 5 is the Gnd pin. This pin is connected to the Gnd contact of the smart card in order to ensure that voltage references for Gnd are all the same. This ensures that the voltage conversion in the MAX232CPE chip would be correct.

RTS (Pin 7) and CTS (Pin 8)

RTS is the Ready To Send pin where else CTS is the Clear To Send pin. These are handshaking signal where whenever one entity wants to sent some data to the other, it would assert the RTS line. If the other entity is ready to receive the data, it would then assert the CTS line indicating that the data can be sent. Since, handshaking is not implemented; these two lines have been connected together. They do not really serve a purpose, as no data will be transferred out of the serial port of the processing module.

4.2.3 PALM Snooping Program

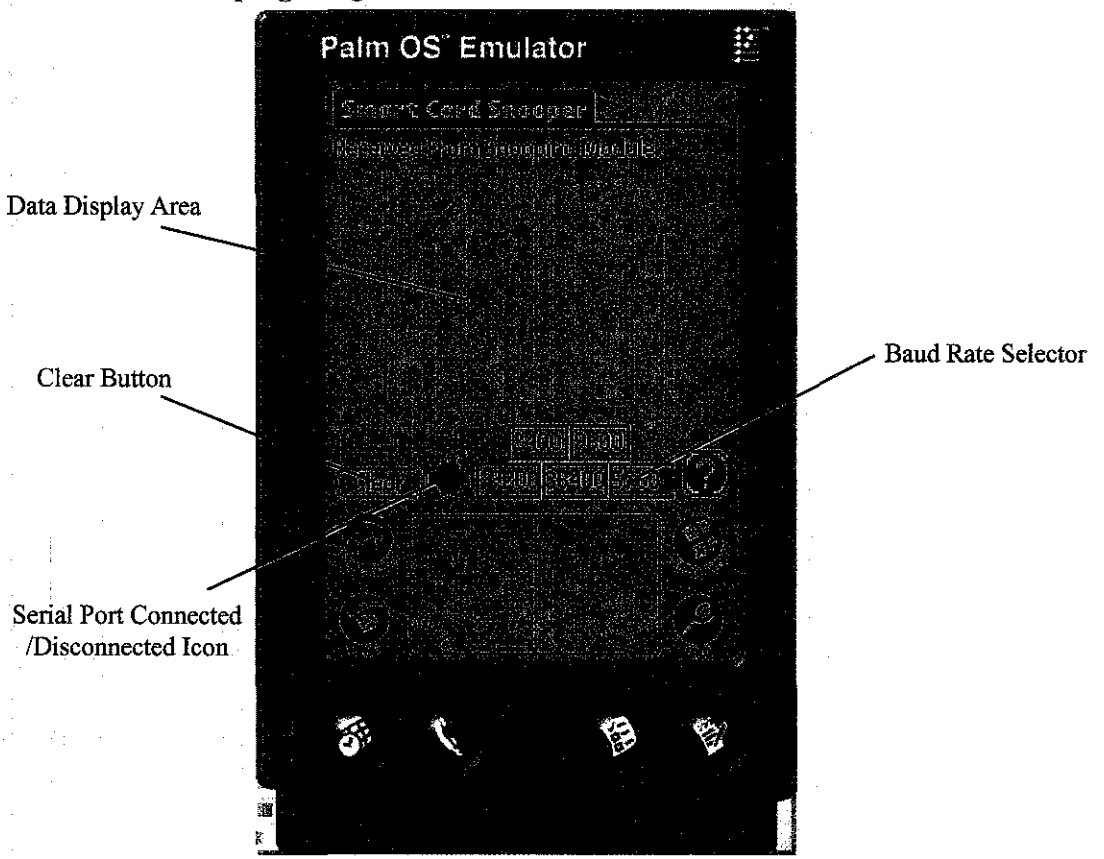


Figure 4.6: Screen shot of the PALM snooping program

The PALM snooping program is the most important component in the processing module. The program is in charge of establishing a connection with the serial port, reading the data, converting it into useful information and finally displaying the information in a user friendly manner.

4.2.3.1 Palm Program Development

The software development process of Palm snooping program involved a lot of groundwork before an actual application was created. The softwares used in order to do the development were custom made for Palm Developement. The CodeWarrior program was the main development software that contained all the source code for the application. The Graphical Interface of the Palm required the use of a separate software called the "Constructor" that will be manually linked to the CodeWarrior. Debugging the Palm applications also was a unique process as it involved special techniques and the use of special programs.

On the actual programming aspect, the method of in order to create the application was also special as Palm application are event based applications that run in an infinite loop. Basically, upon execution of the program, the application is set up and the looped using a while command. In the loop several handlers are available and each of these handlers performs certain functions. When anything is done to the Palm (a button is pressed, something is written or etc.) an event is dispatched into the loop and the event handler will take the appropriate action in order to respond to the incoming event. Once the event has been taken care off, it is discarded and the program goes back to looping till the next event is dispatched into the loop. A typical event loop and its handlers is as the following:-

```
static void EventLoop (void)
{
        Word error;
        EventType event;
        do
        {
                EvtGetEvent (&event, evtWaitForever);
                PreprocessEvent (&event);
                if (! SysHandleEvent (&event))
                if (! MenuHandleEvent (NULL, &event, &error))
                if (! ApplicationHandleEvent (&event))
                FrmDispatchEvent (&event);
        }
        while (event.eType != appStopEvent); // terminate program command received ?
}
```

A complete illustration **on the softwares used and the event loop** for the Palm application is available in Appendix E.

4.2.3.2 Program Overview

Figure 4.6 shows a screen shot of the snooping program. The method of using the program is relatively simple. When the program is launched from the Palm main page, a screen as shown in Figure 4.6 will be seen. At this point the snooping program has already automatically opened the serial port in order to accept data at 9600 baud rate. This was done as 9600 is normally a very common baud rate that is used by smart card readers. If there were incoming data at this point at the rate of 9600 baud, it will be

accepted and displayed in the display area. In order to change the baud rate of the serial port, the user would only have to select one of the 5 baud rates that have been specified by the "baud rate selector buttons". When this is done, the serial port is closed and then re-opened at the new baud rate. Once this happens, the system is once again ready to accept data through the serial port at the new selected baud rate. Once data is accepted and the display area begins to fill up, the "Clear" button can be used in order to clear the screen. When the menu item is selected, the "Close Serial Port Option" and "Open Serial Port" functions are available in order to close or open the serial port manually. On the other hand, in order to exit the program, the user would only have to select anywhere outside the application as the application will automatically close the serial port and exits the program.

4.2.3.3 Receiving the Bytes

When the program is launched, the serial port is automatically opened by launching the port initialization codes at the same time. These initialization codes are basically built in functions and structures that are obtained from the *SerialMgr.h* header file.

In order to connect to the port, certain details regarding the type of transmission, the baud rate, number of start bits, stop bits and type of parity has to be configured. All these tasks have been made easy by an in-built class called the *SerSettingsType* structure. The following is a snippet of the coding from the C++ program in order to declare the required configurations for the serial port.

```
SerSettingsType     SerCommSettings;
CommFlags    =  serSettingsFlagBitsPerChar8|serSettingsFlagStopBits2|
                serSettingsFlagParityEvenM;
                        //specify 8 characters per bit,2 stop bits, Even parity
                        //Comm flag is also an in-built structure
```

In order to implement the coding to connect to the port, certain functions have to be called. The following is the code snippet that connects to the serial port and sets the attributes by declaring the *SerSettingsType* structure that was created above.

```
#define PORT                    0
error = SysLibFind("Serial Library", &gSerRefNumber);
error = SerOpen(gSerRefNumber, PORT, Baud);
                    //Open Serial Port at specified  baud, PortID returned to
                   //gSerialPortID

SerCommSettings.baudRate =  Baud;                    //Set th specified baud rate
SerCommSettings.flags = CommFlags;                   //Set flags
error = SerSetSettings(gSerRefNumber, &SerCommSettings);
                                                     //Set communication settings
```

The *SysLibFind* is a function used to get the reference number for a library. Since "Serial Library" was specified as the parameter in the command, this function returns the serial number for the serial library. This number is used in order to open the serial port (*SerOpenPort*). As can be seen, the *SerCommSetting* is the structure that contains all the information regarding the specifications of the serial port connection. After opening the port, these settings were applied to the serial port using the *SerSetSettings* function.
Appendix K provides a brief overview on the ready-made function and structures that have been used in order to create the program.
Besides the programming that has been done to achieve the successful connection and data transfer, it is also important to acknowledge the fact the serial port contains a UART that performs the transfer of serial bits to a single byte. In this case, the UART in the serial port reforms the bytes by striping away the overhead bits that have been added.

4.2.3.4 Bytes conversion subroutine

The byte conversion subroutine is the portion of the program that functions to convert the bytes received into its ASCII representation of the number in order to properly display it. For instance if a bytes 0x00 is received, in order to display 0x00, it has to be converted in to 0x30 and 0x30 where 0 is represented as hex 30 in ASCII. The data is received by periodically checking the function

```
error = SerReceiveCheck(gSerRefNumber, &NumberOfBytes);
        //Retrieve the number of bytes to be retrieved
   while (NumberOfBytes)                          //Retrieve data
        {
        SerReceive(gSerRefNumber, &chrArray, 1, 0, &error);
        //Retrieve byte one at a time into chrArray

        if (error == serErrLineErr) SerClearErr(gSerRefNumber);
                                //Clear Error on Line Error
        if (error)
                {
                SerReceiveFlush(gSerRefNumber, 1);
                        //Clear buffer on error
                        i = 0;                                  //Reset Array Index
                        NumberOfBytes = 0;
                        return;
                }
        else
                {
                        i++;                                    //Increase Array Index
                        NumberOfBytes--;        //Decrease number of bytes to be retrieved
                        FieldP = GetObjectPtr(Main_ReceivedFromExternalField);
                                        //Get Pointer to Field
                        StrCopy(&gMessageToDisplay[0], &chrArray);
                                        //Copy into display string
                        StrCopy(&gMessageToDisplay[1], "");
                                        //Terminate with NULL
                        ConvertMessageToDisplay();
                                        //Convert character to hex if non-printable
                        DisplayData(FieldP);
                                        //Display Data into above Field
                         UpdateScrollBar () ;
                        chrArray = '/0';                //Reset array
                        tmpBuffer = '/0';               //Reset buffer
                }
        }  // end while loop
```

When the *SerReceiveCheck* function returns a value through the *NumberOfBytes* parameter, this means that data is available. Upon receiving an indication that there is incoming data, the program then enter a while loop that will last for the same number of cycles as the lengths of the incoming bytes. *SerReceive* is the function that is used in order to receive the bytes one by one. The reasons only one byte is received at one go is

because this was specified in the *SerReceive* function. After receiving the data (one byte), it is then processed so that it will be converted into its ASCII representation. The function that does this is the *ConvertMessageToDisplay* function.

```
void ConvertMessageToDisplay()
{
Int     DecNumberToConvert;
DecNumberToConvert = (int)((unsigned char)(gMessageToDisplay[0])); //Get character
ConvertToHex(DecNumberToConvert);                                  //Convert to hex
}
```

This function in turn gets the equivalent decimal value of the incoming data. The decimal value is then sent into another function, the *ConvertToHex* function. This function was created in order to convert the bytes to its ASCII representations. Given below is the snippet of the code illustrating how the *ConvertToHex* function actually transforms the bytes into its ASCII representations[14].

```
void ConvertToHex(Int NumToConvert)
{
        UInt    tmpNumber = 0;
        UInt    counter = 1;
        UInt    i = 1;

        StrCopy(gMessageToDisplay, "");                    //Clear String
        do
        {
        if (counter == 1)
                (tmpNumber = NumToConvert/16);  //Calculate upper nibble
        else
        (tmpNumber = NumToConvert - (tmpNumber * 16));//Calculate lower nibble
                switch (tmpNumber)
        //Add appropriate values
                {
                case 0:
                        StrCat(gMessageToDisplay, "0");
                        break;
                case 1:
                        StrCat(gMessageToDisplay, "1");
                        break;
```

[14] refer to Appendix I in order to view the ASCII table.

```
case 2:
        StrCat(gMessageToDisplay, "2");
        break;
case 3:
        StrCat(gMessageToDisplay, "3");
        break;
case 4:
        StrCat(gMessageToDisplay, "4");
        break;
case 5:
        StrCat(gMessageToDisplay, "5");
        break;
case 6:
        StrCat(gMessageToDisplay, "6");
        break;
case 7:
        StrCat(gMessageToDisplay, "7");
        break;
case 8:
        StrCat(gMessageToDisplay, "8");
        break;
case 9:
        StrCat(gMessageToDisplay, "9");
        break;
case 10:
        StrCat(gMessageToDisplay, "A");
        break;
case 11:
        StrCat(gMessageToDisplay, "B");
        break;
case 12:
        StrCat(gMessageToDisplay, "C");
        break;
        case 13:
        StrCat(gMessageToDisplay, "D");
        break;
case 14:
        StrCat(gMessageToDisplay, "E");
        break;
case 15:
        StrCat(gMessageToDisplay, "F");
        break;
#if(debug)
default:
        StrCat(gMessageToDisplay, "0");
        break;
```

```
            #endif
            }//End Switch
        counter--;
        i++;
        } //End While
        while (counter != -1);
}  //End Function ConvertToHex
```

4.2.3.5 Data Interpretation

The data obtained from the snooping device via the PALM C++ program is very meaningful. As explained in section 4.2.1, there are basically two main types of data formats that are used, the Answer to Reset (ATR) and the Application Protocol Data Unit (APDU).

4.2.3.5.1 ATR Standard Format

The ATR is a string of characters returned from the card indicating a successful power-up sequence. The total length of the ATR sequence is limited to 33 bytes and must adhere to the following format:

Name	Number	Function	Presence
TS	1	Initial Character	Mandatory
T0	1	Format Character	Mandatory
TA_i, TB_i, TC_i, TD_i	<15	Interface Characters	Optional
T1, T2...TK	<15	Historical Characters	Optional
TCK	1	Check Characters	Conditional

Table 4.2: The Answer-To-Reset structure

TS and T0 are the only mandatory bytes in the ATR sequence. As previously described, the initial character TS is used to establish bit-signaling and bit-ordering conventions. T0 is used to indicate the presence or absence of subsequent interface or historical

characters. The upper 4 bits (bits 5 - 8) are designated Y1 and signals the presence of optional characters based on a logic 1 in the following bit positions:

- Bit 5 indicates TA1 is present
- Bit 6 indicates TB1 is present
- Bit 7 indicates TC1 is present
- Bit 8 indicates TD1 is present

The lower 4 bits (bits 1 - 4) are designated K and is interpreted as a numeric value in the range 0 - 15. It indicates the number of historical characters present.

The interface characters are used to select the protocol used for subsequent higher-level communication between the smart card and the reader. ISO 7816-3 defines two protocols: the T=0 protocol and the T=1 protocol. T=0 is an asynchronous *character-oriented protocol* where an acknowledgement must be received for every byte that is sent. In contrast, T=1 is an asynchronous *block-oriented protocol* where a number of bytes can be sent before an acknowledgement must be received.

The historical characters are usually used to indicate the type, model and use of the specific card. The manufacturer or card issuer generally defines these. There is no established standard for the data in these historical bits. The check character (TCK) used to determine whether a transmission error occurred in sending the ATR from the card to the reader. TCK is a checksum calculated such that performing a bit-wise exclusive-or (XOR) operation on all bytes in the ATR from T0 to TCK results in an answer of zero.

Refer to Appendix J for a more detailed table of the ATR.

4.2.3.5.2 APDU Standard Format

The APDU is a string of bytes that have been formatted according to a standard format that make up a command to the smart card. In return, the card will also always reply with its own standardize response. This response indicates the success or failure of the command and may also have information attached to it if requested by the command.

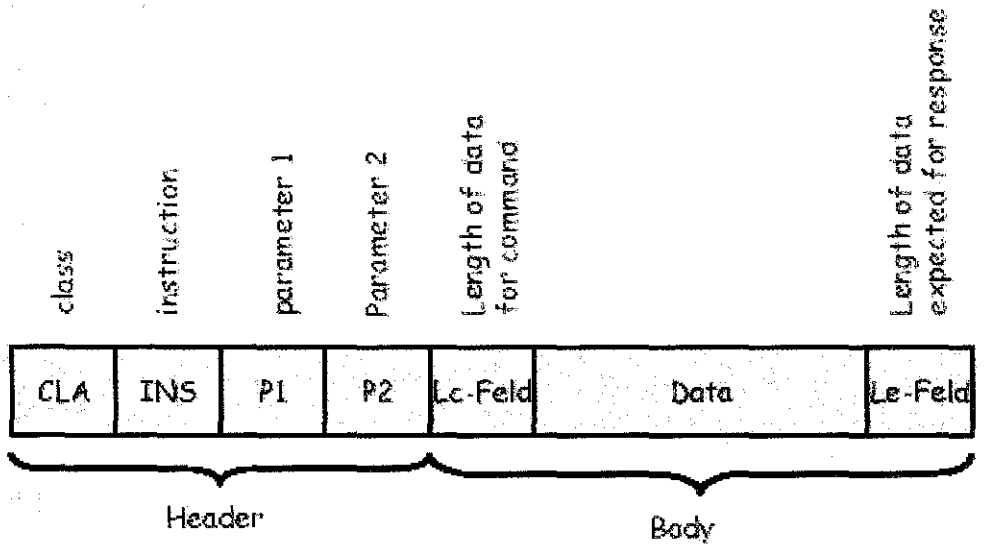


Figure 4.7: APDU format

Figure 4.7 shows the basic format for the APDU. The 5 bytes shown at the beginning are called the header of the APDU. The CLA and INS bytes are indications of the type of command that is being invoked. Bytes P1 and P2 on the other hand are bytes that contain the setting for certain options that are related to the command. The L_c field indicates the number of bytes that would be trailing the APDU header. The data portion is filled up with the necessary information if there is anything to be sent to the card .The final byte, the L_e field is also optional as the use of it depends whether or not the card reader expects data to be transmitted from the card.

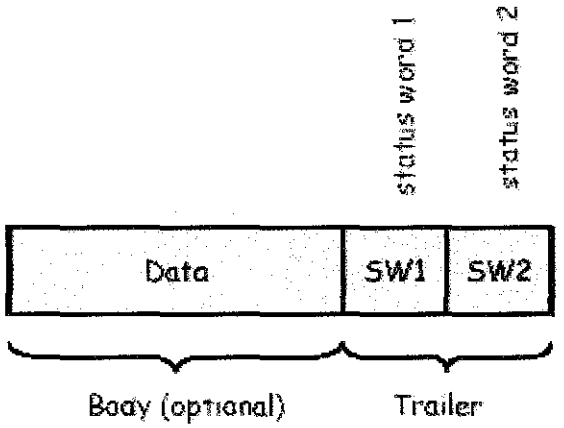


Figure 4.8: APDU response format

Figure 4.8 shows the standard APDU response format. For every command (APDU) sent to the card, the SW1 and SW2 (which are known as the *status word)* are two compulsory bytes that must be sent from the smart card to the reader after every command. These are the two bytes that indicate the success or failure of the command. Figure 4.9 gives a brief overview on the possible values of the return code.

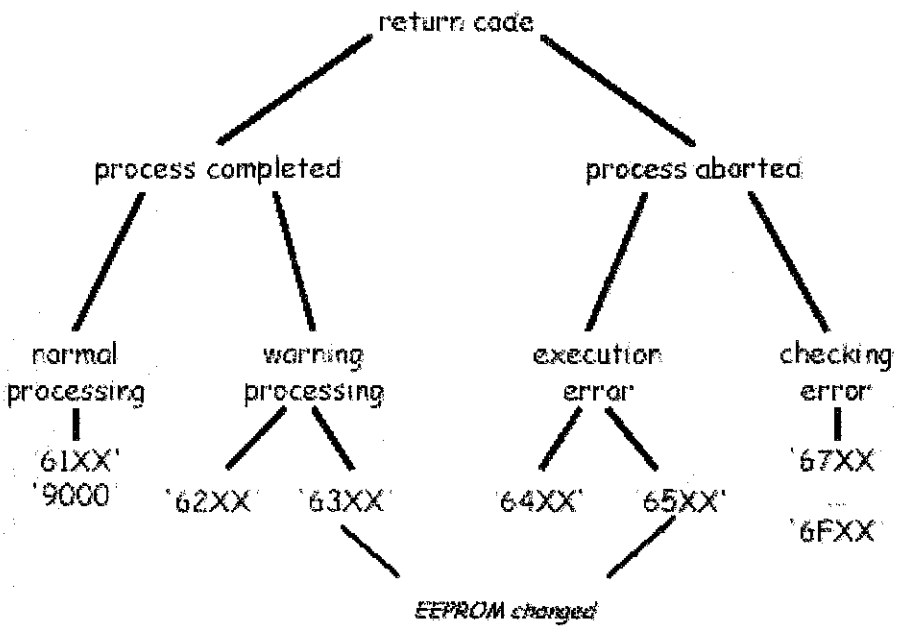


Figure 4.9: Classification scheme for the APDU return code (SW)

The data field that precedes the SW field in Figure 4.8 is the data that will be sent from the card (if the command succeeds) to the reader. The length of this field is stated in the APDU in the Le field.

A sample of the data log obtained from the snooping device can be seen in the following section. The data transmission between the card and the smart card reader is briefly discussed in this section in order to give a better understanding of the log based on the information that has been presented above.

4.3 ELABORATION ON THE FINAL OUTPUT

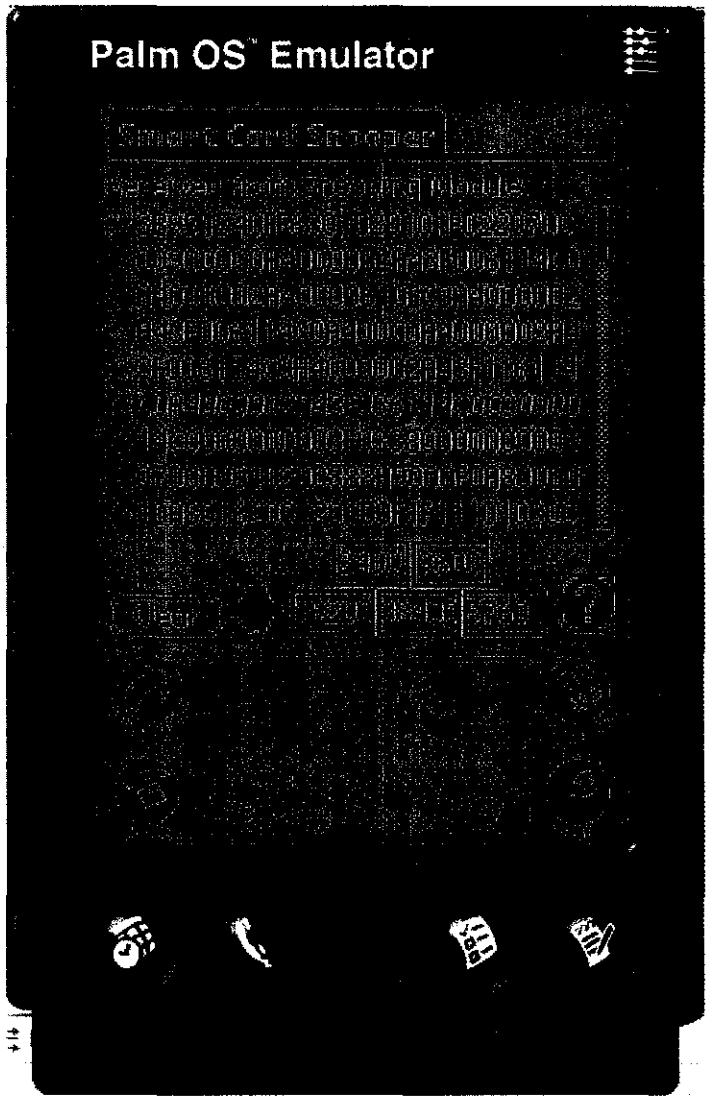


Figure 4.10: Sample log that was obtained from the snooping device program during the data transfer

Figure 4.10 is a sample log that is obtained during the insertion of a smart card into the smart card reader. The text Palm emulator displays a portion of the data communication that took place (first few lines). There are many more commands that were executed at the bottom of the display. However, for the sake of illustrating the data transmission, focus will be given to the first few lines.

As stated in the report the transmission normally begins with the card transmitting its ATR. This is followed by the PTS if necessary and the rest of the commands are basically APDU's and responses.

The following is an exact copy of the data in Figure 4.10 with the lines numbered for the ease of illustration.

Line 1 : 3B951840FF6201020104 F0220700

Line 2 : 009000 C0A4000002A43F006114 C0

Line 3 : A4000002A40000610FC0A4000002

Line 4 : A43F006114 C0A400C0A4000002A4

Line 5 : 3F006114 C0A4000002A43F116114

Line 1(beginning) is the first transmission that is from the card. As stated the first transmission is always the ATR. Hence 3B951840FF6201020104 is actually the ATR of the card. Elaboration on the meaning of each byte can be obtained from Appendix L and *section 4.2.3.5.1.* In order to verify that those bytes were actually the ATR, certain smart card software was used in order to check the ATR.

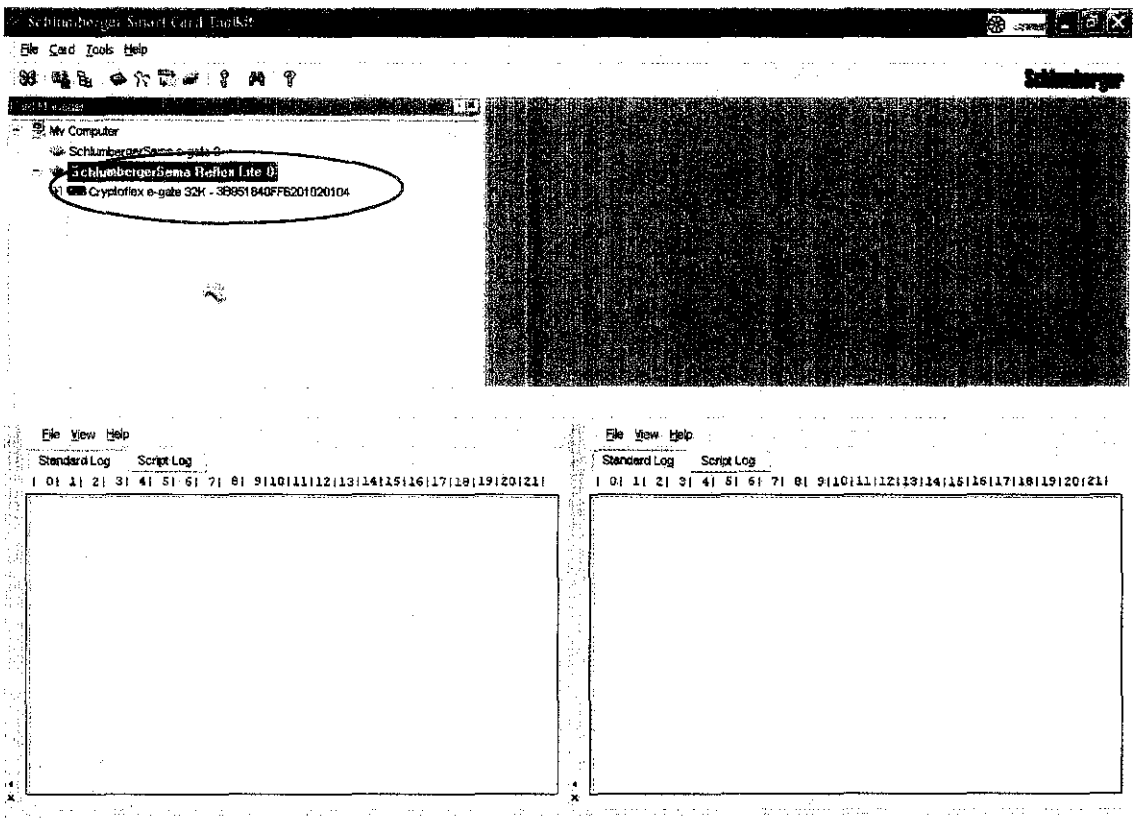


Figure 4.11: Image smart card software used (Schlumberger Smart Card Toolkit)

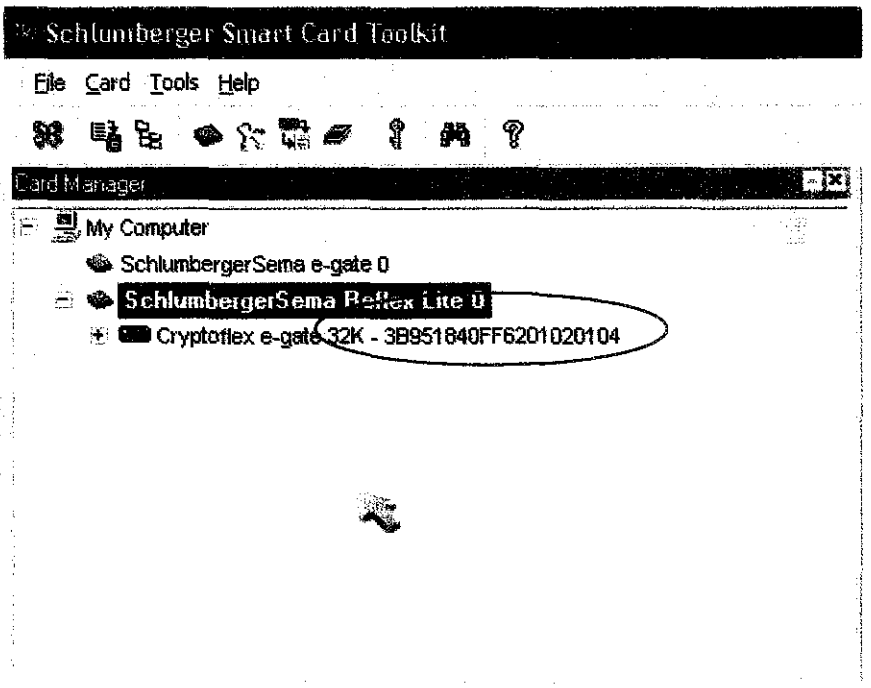


Figure 4.12: Taking a closer look into the program screen shot (area marked in red)

Figure 4.11 shows the smart card software that was used, the Schlumberger Smart Card Toolkit. When the card has been inserted, the ATR of the card appear at the portion marked in red. A closer look at that portion is shown in Figure 4.12. Note that the ATR in the software and the ATR in Line 1 is **exactly the same.**

As stated in *section 4.2.1*, after the transfer of the ATR bytes, the programs proceeds with the command (APDU) and response sequence as most of the time the PTS bytes are not needed. In the log shown, the PTS bytes are not needed as Line 4 onwards basically corresponds to the APDU format.

The bytes in Line 1(ending) and 2 (beginning) are F0220700009000. Breaking the bytes into two portions, we can see that the command APDU is actually F022070000 where else the response status word (SW) is 9000. Refer to figure 4.9 to check up on the returned status word.

Command APDU					Response SW	
CLA	**INS**	**P1**	**P2**	**L_c**	**SW1**	**SW2**
F0	22	07	00	00	90	00

Table 4.3: Interpretation of Line 1(Ending) and Line 2 (beginning)

The data in Line 2 (middle) (C0A4000002A43F006114) can also be classified using the same manner as stated for the command in Line 1(ending) and 2 (beginning) above. However, there are certain differences between Line 2 (middle) and Line 1(ending) and 2 (beginning). First of all, since the L_c field is 02, this means that there are two data bytes that will trail the APDU header. The second difference is that sometimes, there are some extra bytes that are transferred to the smart card in order to act as a separator. These bytes are not actually part of the data packet.

Command APDU							Response SW	
CLA	**INS**	**P1**	**P2**	**L_c**	**Data**		**SW1**	**SW2**
C0	A4	00	00	02	3F	00	61	14

Table 4.4: Interpretation of Line 2 (middle)

From the table above, it can be seen that there is one byte that was missed out, the A4 byte after the L_c parameter. This is the special byte that acts as a separator. By referring to Line 3,4 and 5, it can be seen that this byte and probably a C0 byte is repeated in each and every APDU transfer. The status words that are returned are rather straightforward.

All the communication from line 2 onward (after the ATR in line 1) complies with the standard APDU format. Hence, each and every APDU send to the card can be logged and the response sent from the card to the reader can also be documented.

This repeats until the card is disconnected or reset. If the card is reset, the data transmission starts all over again with the transmission of the ATR and the corresponding APDU's as shown in Figure 4.10.

CHAPTER 5

RECOMMENDATION & CONCLUSION

5.1 RECOMMENDATION

5.2 CONCLUSION

CHAPTER 5
RECOMMENDATION & CONCLUSION

5.1 RECOMMENDATION

The device created was successful as it is able to effectively snoop on the communication between a smart card and the smart card reader. The snooping device is portable and is able to functions at various speeds depending in the smart card reader. Nevertheless, there are still a few modifications that could be made to the device in order to further improve its functionality.

1. **Integrate all the snooping and communication modules into a single device that can be neatly connected to the processing module, the Palm.**

   The idea that could be used in order to do this would probably be do create a box like device where all the circuit could be placed into it. This box would probably have a slot to connect to the Palm and another slot to accept actual smart card. The dummy smart card can be extended from this box and connected to the actual smart card.

2. **Improve the snooping program in order to be able to not only accept and format the data but also interpret and differentiate between the ATR, standards smart card command and the returned status word.**

   Currently the snooping program accepts the transmitted data and displays it on the Palm device. The data displayed is complete but the method as to which it is displayed could be made easier. Each function and its retuned status word are recorded sequentially without any differentiation between them in the display. If the program could read and make sense of the bytes , it would be possible to format the data so that it displays the commands as a set.

   This can be done by referring to the IS0 7816 standard. Since the format has been clearly illustrated, it should be possible for the program to process he data at a higher level.

3. **Convert the device in order to be able to connect to the USB port also.**

   This would be a good idea in order to make the data transfer rate faster hence ensuring that the snooping device does not miss out on any data bytes.

5.2 CONCLUSION

The main objective of the project was to design a "chip based smart card" snooping device that was both portable and affordable. This device was successfully designed and implemented using various devices and methods that have been worked on for the entire duration of the project.

The final design for the project involved a hardware configuration that can be classified into three modules. The first module, the snooping module is a combination of two specially engineered components that are used in order to tap on the required signal lines between the card and the reader. The second module, the communication module functions to encode the signals into the required voltage levels and format for the serial port. Lastly, the processing module accepts the data through the serial port, processes it and displays the byte exact byte transfer in the screen.

The processing module for the final design was a PALM device that was an affordable and more importantly, portable device. In addition, an initial prototype that implements the PC as the processing module is also available as this was a preliminary design for the project. In both designs, the data log that was created in the processing module complies with the ISO-7816 smart card communication standards. This standard is the basic byte transfer format for the communication between the smart card and the smart card reader. Hence, this verified the design of the devices and ensured confirmed the reliability of the obtained result.

All in all, the all the project objectives have been achieved and the importance and significance for a device such as the Universal "Chip Based Smart Card" Snooping Device in smart card related implementations has been entirely fulfilled.

CHAPTER 6

REFERENCES

CHAPTER 6
REFERENCES

Books

[1] William Stallings, **Data and Computer Communication**, Sixth Edition 2003,Prentice Hall International Inc.

[2] Henry Dreifus, **Smart Cards:A guide to building and managing smart card applications** , 1st Edition 1996, John Wiley & Sons Inc.

[3] Jose Luis Zoreda, **Smart Cards**, 1st Edition 1994, ARTECH HOUSE Inc.

[4] Robert MykLand, PALM OS Programming from the Ground Up, 1st Edition 2000, McGraw-Hill

[5] Steve Mann, Advanced PALM Programming, 1st Edition 2001, John Wiley & Sons Inc.

[6] Eric Giguere, Palm Database Programming, 1st Edition 1999, John Wiley & Sons Inc.

[7] Jose Luis Zoreda, **Smart Cards**, 1st Edition 1994, ARTECH HOUSE Inc.

[8] PALM OS SDK Documentation

Softwares

[9] Microsoft Software Developers Network (MSDN) October 1999

[10] Microsoft Visual C++

[11] CodeWarrior Lite for PALM OS

[12] PALM OS Emulator

Websites

[13] www.codeproject.com

[14] www.codeguru.com

[15] http://www.epanorama.net/links/pc/interface.html#serial

[16] http://www.maxking.co.uk/iso78161.htm

[17]http://freebsd.unixtech.be/doc/en_US.ISO8859-1/articles/serial-uart/

[18]http://www.byterunner.com/**16550**.html

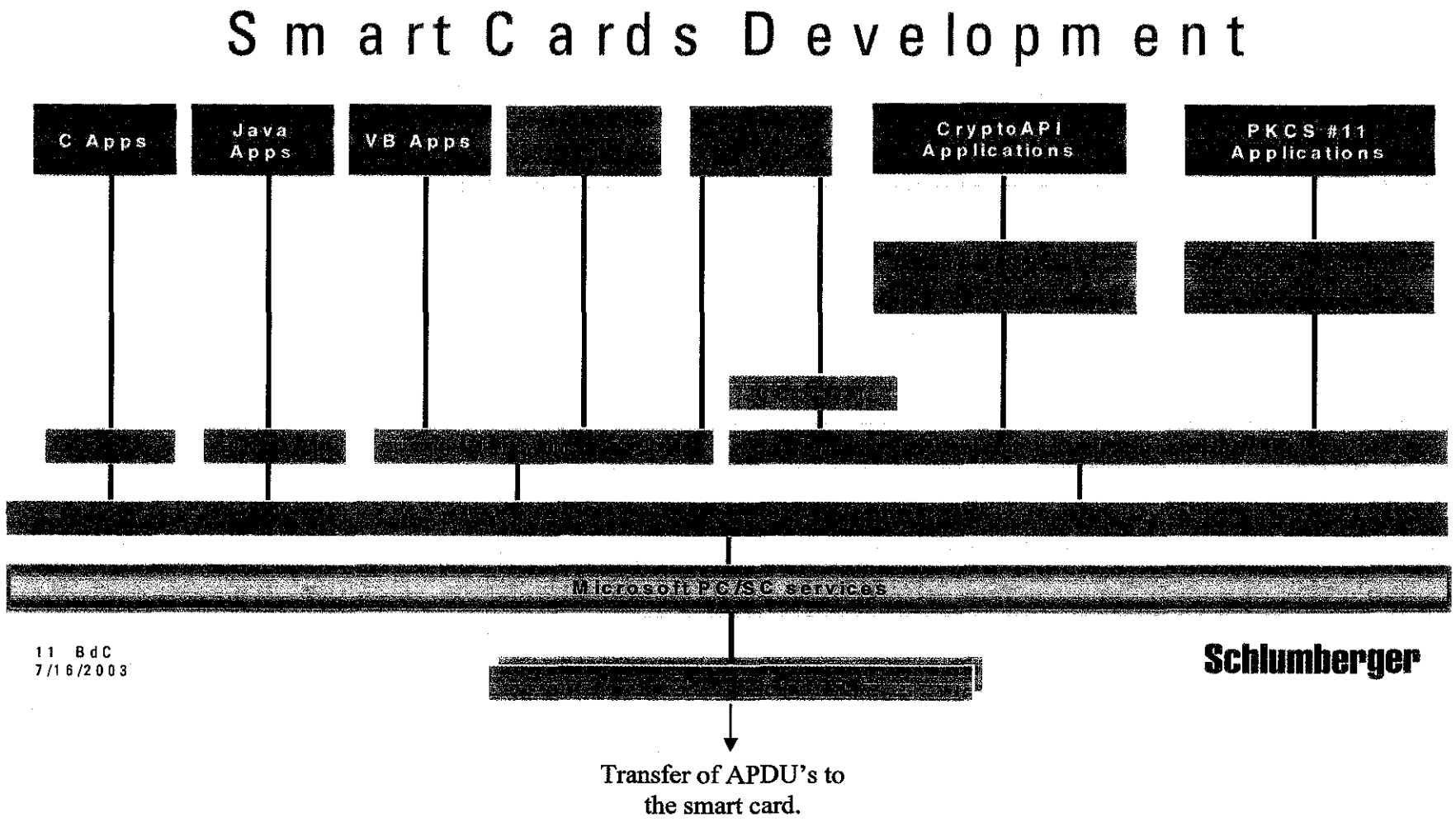
[19]http://www.activxperts.com/activcomport/progruart/

[20] http://www.palmos.com

Appendix A

Smart Card Development Diagram

APPENDIX A: Smart Card Development Diagram



Appendix B

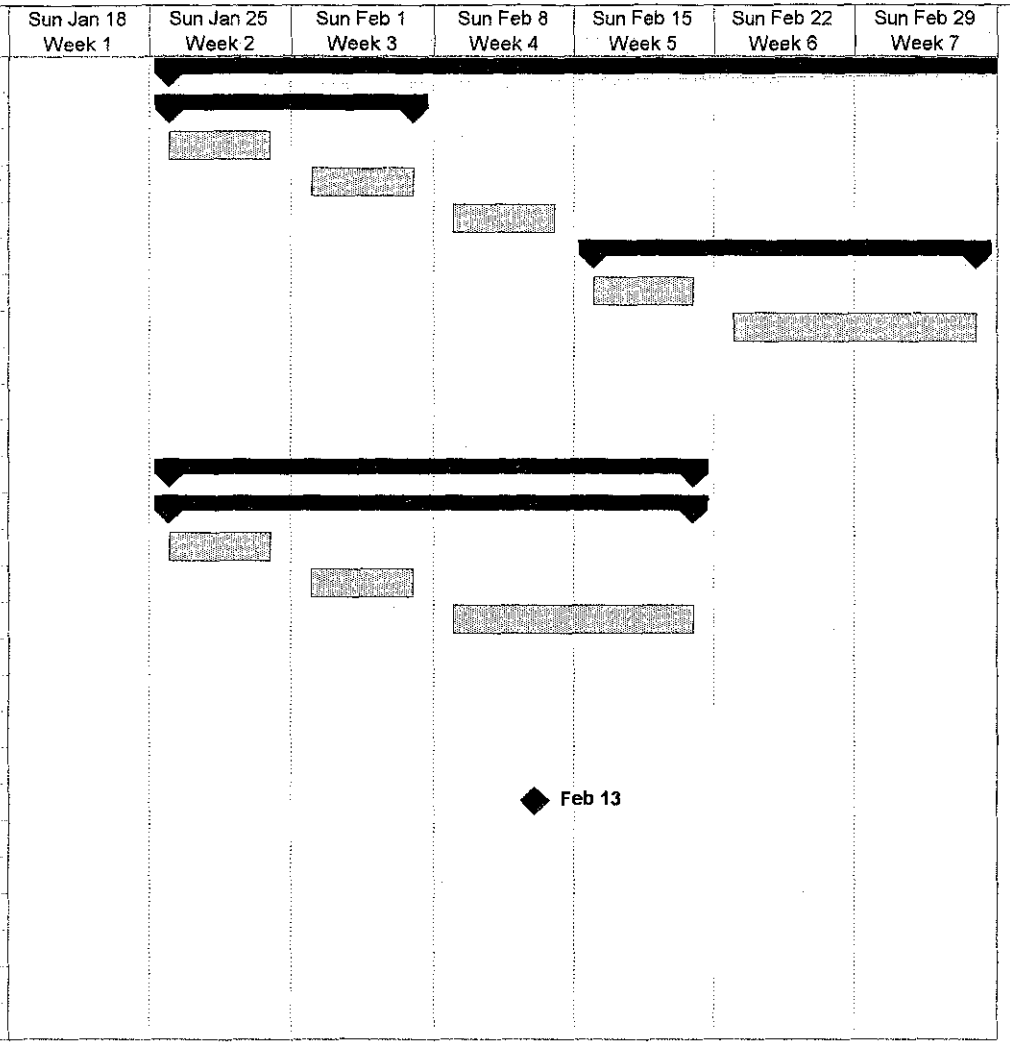
Project Gantt chart

APPENDIX B: Project Gantt chart for 1st half of Project

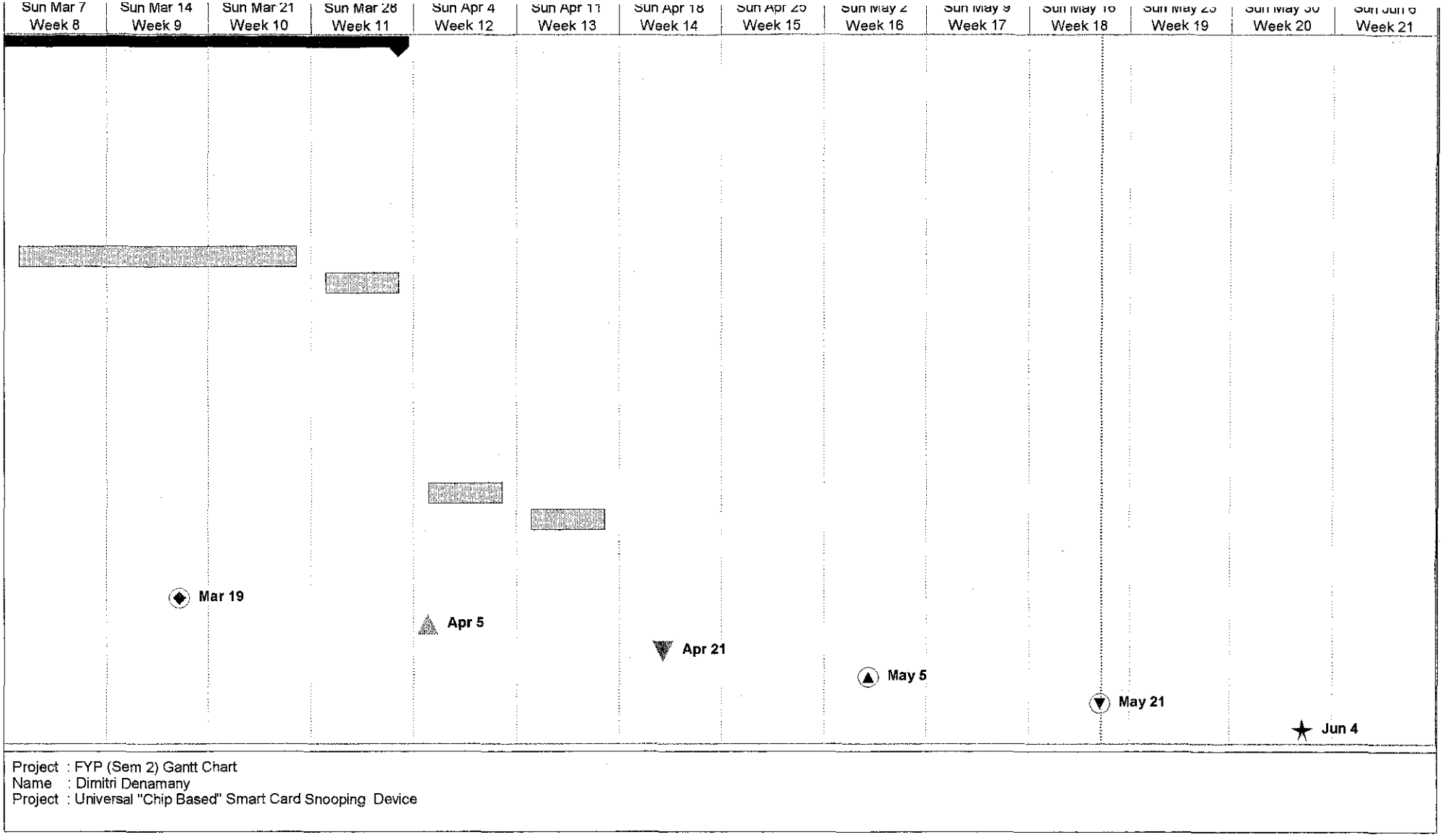
No.	Detail/ Week	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	Selection of Project Topic														
2	Submission and Conformation of Topic														
3	Preliminary Research Work														
	- Possible Hardware Designs Creation														
	- RS232 Components Identification														
4	Submission of Preliminary Report														
5	Testing and conformation of hardware design														
6	Research Work														
	- Smart Card Communication														
	-UART														
7	Research Work														
	- MAX232														
	- COM port -RS232 communication (VC++)														
8	Submission of Progress Report														
9	Testing card input through RS232 (VC++)														
10	Creation of the software to interpret the data														
11	Hardware manipulation and integration														
12	Submission of Interim Report Final Draft														
13	Submission of Interim Report														
14	Oral Presentation														

- 15 August 2003
- 12 September 2003
- 10 October 2003
- 22 October 2003
- TBA

ID	Task Name	Duration	Start	Finish
1	**Processing Module**	**51 days**	**Mon 1/26/04**	**Fri 4/2/04**
2	**Preliminary Research**	**10 days**	**Mon 1/26/04**	**Fri 2/6/04**
3	Microcontrollers	1 wk	Mon 1/26/04	Fri 1/30/04
4	Palm Devices	1 wk	Mon 2/2/04	Fri 2/6/04
5	Logging Device Selection	1 wk	Mon 2/9/04	Fri 2/13/04
6	**Research Work**	**15 days**	**Mon 2/16/04**	**Fri 3/5/04**
7	Software and Hardware Identification	1 wk	Mon 2/16/04	Fri 2/20/04
8	Programming Study	2 wks	Mon 2/23/04	Fri 3/5/04
9	Processing Module Programming	3 wks	Mon 3/8/04	Fri 3/26/04
10	Software Testing	1 wk	Mon 3/29/04	Fri 4/2/04
11				
12	**Snooping & Communication Module**	**21 days**	**Mon 1/26/04**	**Fri 2/20/04**
13	**Hardware Design and Fabrication**	**21 days**	**Mon 1/26/04**	**Fri 2/20/04**
14	Dummy Smart Card	1 wk	Mon 1/26/04	Fri 1/30/04
15	Connector Module	1 wk	Mon 2/2/04	Fri 2/6/04
16	Hardware Integration	2 wks	Mon 2/9/04	Fri 2/20/04
17				
18	Modules Integration	1 wk	Mon 4/5/04	Fri 4/9/04
19	Testing and Troubleshooting	1 wk	Mon 4/12/04	Fri 4/16/04
20				
21	Progress Report 1	0 days	Fri 2/13/04	Fri 2/13/04
22	Progress Report 2	0 days	Fri 3/19/04	Fri 3/19/04
23	Draft Report	0 days	Mon 4/5/04	Mon 4/5/04
24	Final Report (Soft Cover)	0 days	Wed 4/21/04	Wed 4/21/04
25	Oral Presentation	0 days	Wed 5/5/04	Wed 5/5/04
26	Extended Abstract	0 days	Fri 5/21/04	Fri 5/21/04
27	Final Report (Hard Cover)	0 days	Fri 6/4/04	Fri 6/4/04



Project : FYP (Sem 2) Gantt Chart
Name : Dimitri Denamany
Project : Universal "Chip Based" Smart Card Snooping Device



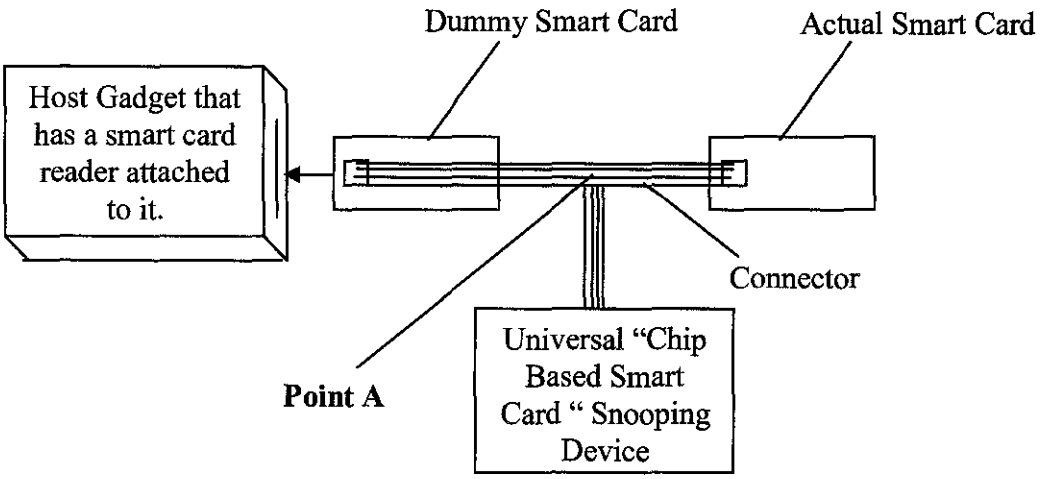
Project : FYP (Sem 2) Gantt Chart
Name : Dimitri Denamany
Project : Universal "Chip Based" Smart Card Snooping Device

Appendix C

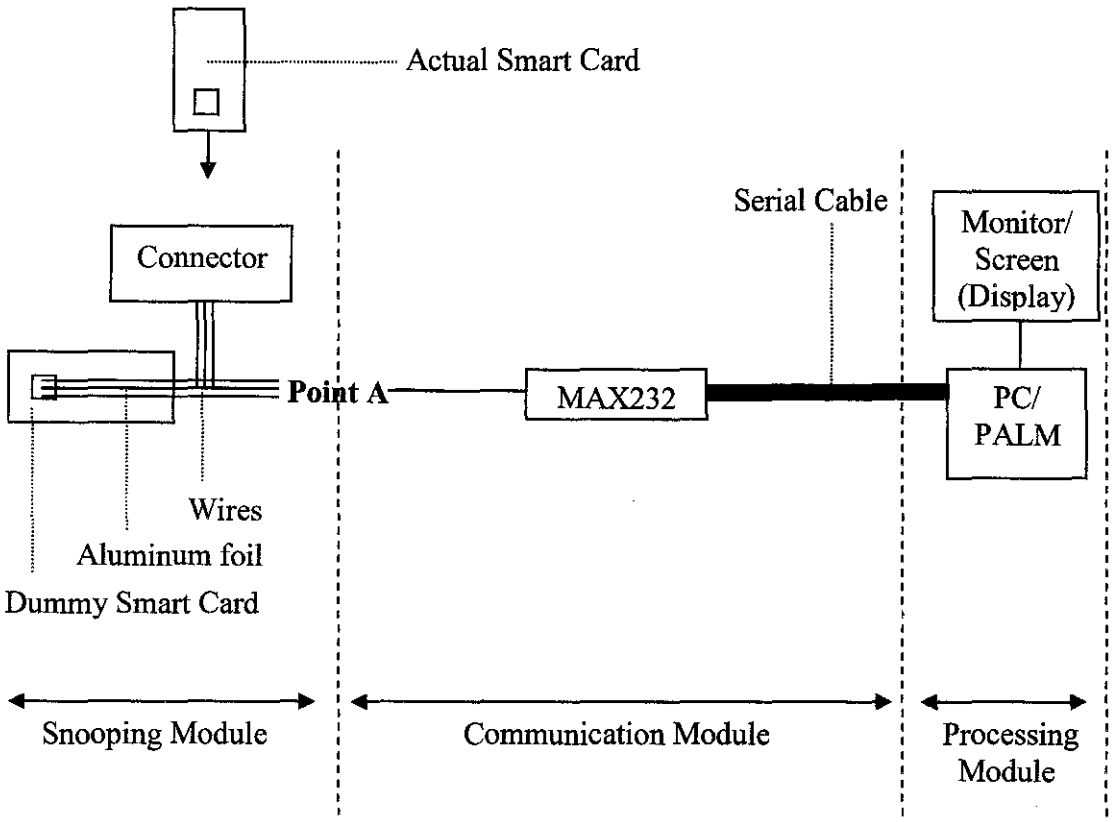
Final Design Specification

APPENDIX C: Final Design Specification

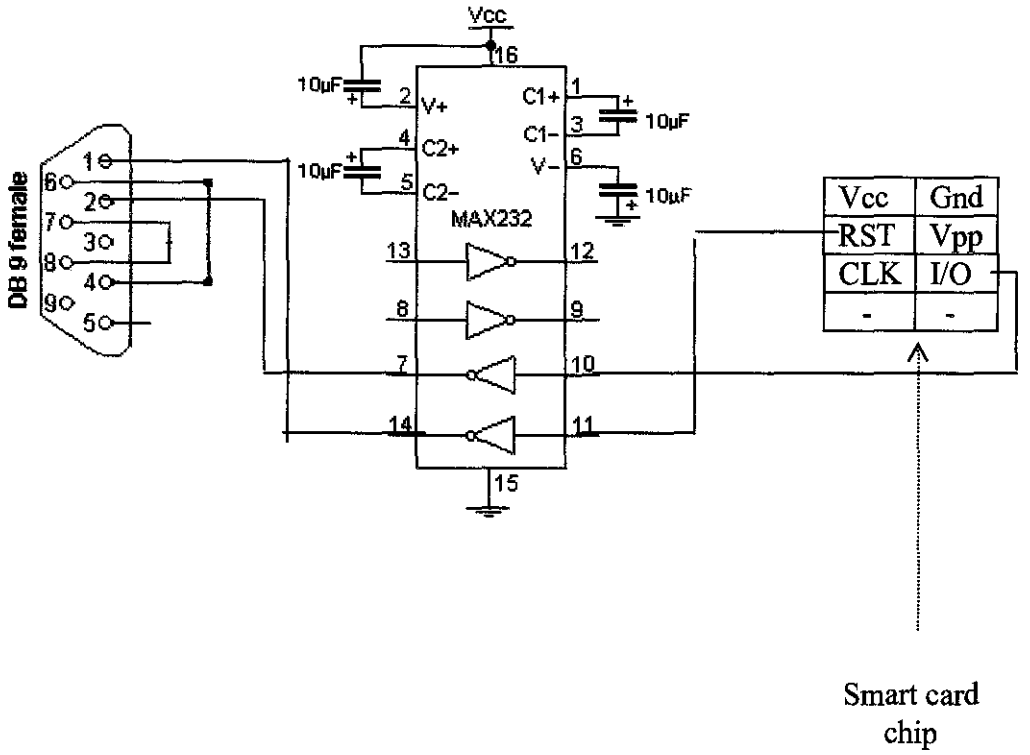
General Block Diagram of Hardware Design



Detailed Block Diagram of Hardware Design (First and Final prototype)



Schematics of the connection from the MAX232 to the RS232 (DB9) cable

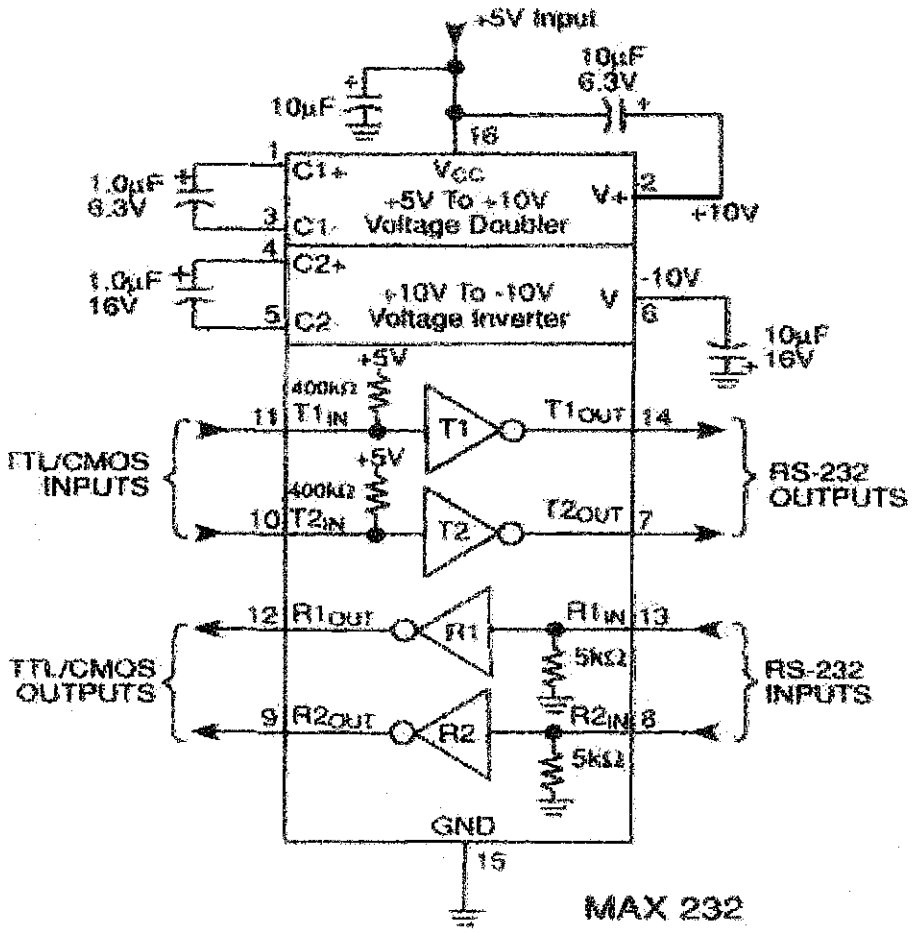


Appendix D

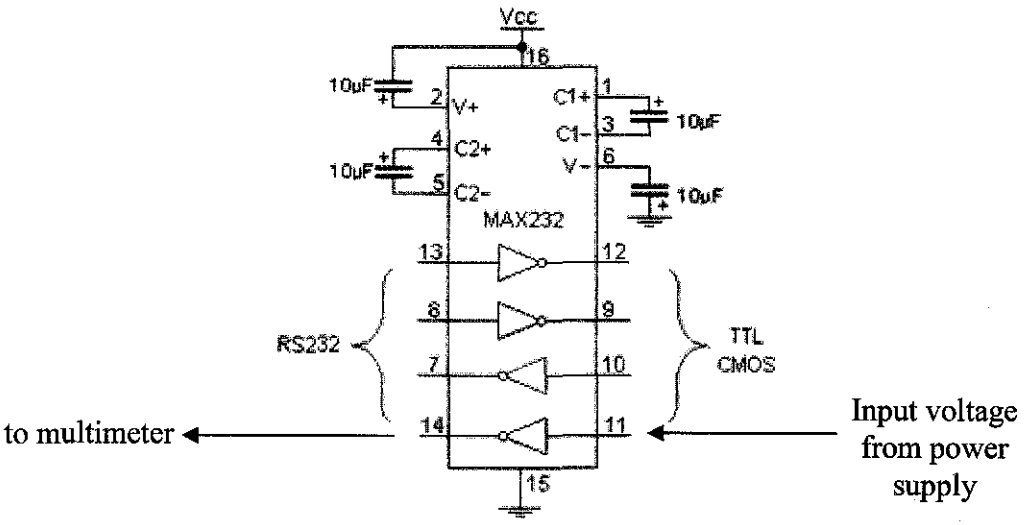
MAX232 Test Circuit Schematic

APPENDIX D: MAX232 Test Circuit Schematics

MAX232 Chip Pin out



Test Circuit



In order to test the chip, a high (5V) input and a low (0V) was inserted in pin 11. The output voltage was the measured using a multimeter and recorded. The recorded value was then compared against the theoretical value.

Appendix E

PALM Software and Programming Basics

APPENDIX E: PALM Softwares and Programming Basics

E.1. PALM Softwares

After deciding on using the PALM as the processing module, a thorough research was done on the various tools needed for PALM programming. The research identified the following softwares to be the main requirements for programming:-

- PALM OS Software Developers Kit (SDK)
- Code Warrior Lite
- PALM OS Emulator
- PALM Handheld Software

All the above stated softwares were downloaded from online sources. The following is a brief description of each of the softwares

E.1.1 PALM OS Software Developers Kit (SDK)

The PALM SDK is basically a bunch of files that contains the various header (.h) and library (.lib) files that are used in order to program the PALM. These files are needed in order to implements the thousands of predetermined function calls that have been outlined in the PALM Programmers Documentation that comes with the kit. The documentation provided with this install is also very helpful as it is basically a complete guide to PALM programming. In the documentation, there is an implementation of a Serial Manager class which contains the various functions in order to manipulate the serial port of the PALM. This is a good indication that the possibility of the success of the project using the PALM is very high.

E.1.2 CodeWarrior Lite

This software is the main tool used in order to create applications for the PALM. Code Warrior Lite is the target software where the programming for the PALM is done. This software has a built in compiler and debugger that makes it a complete development tool. Even so, the debugger actually needs the emulator (next section) in order to fully operate. Code Warrior also comes with another software called the Constructor. This software

was specially created in order to design the User Interface of the PALM. The files created by the Constructor are then imported into Code Warrior Lite in order to link the programming codes to the User Interfaces.

E.1.3 PALM OS Emulator

The PALM Emulator is a unique software that is designed in order to test and debug applications without affecting the actual PALM Pilot system. The emulator is basically a virtual PALM device that runs on the PC. Hence, in order to test a device, it need not be physically downloaded into the actual PALM but instead it can be tested on the PC itself. In terms of debugging, the Emulator also plays an important role as it can be linked to the Code Warrior debugger. Upon loading the program into the emulator, the debugger is then executed hence allowing the developer to single step through the program via the emulator. The emulator most definitely is a useful tool as it saves the actual PALM from the abuse of corrupt and harmful programs.

The only catch with the emulator is that it requires a .ROM file in order to function. A .ROM file is actually a carbon copy of the Operating system running in the actual PALM. Once obtained, this file is loaded into emulator in order for it to function. There are only 2 ways of obtaining the ROM file for an emulator. The first method is to manually download the file from a PALM device. The second method is to apply for the file online and then mail original signed contracts to the PALM Head quarters at the USA in order to gain access to these files.

E.1.4 Palm Handheld Software

This software is the software that comes together with the PALM device that is being used. The purpose of this software (in regards to this project) is to help download the developed applications into the PALM device.

E.2. PALM Programming

E.2.1 Programming Environment Setup

E.2.1.1 Overall Application Development

As stated in the previous section, the CodeWarior Lite software is the main programming environment that will be used in order develop the source code, and link all the required files in order to output a PALM executable application (.prc). For every PALM application that is developed, a project file has to be created in order to contain all the relevant source files, header files and resources files. The Constructor software on the other hand, is used specifically to create the resource file and save it in the project file. The emulator is the software used in order to run applications on the desktop computer as well as debug them[1]. The following is a simplified step by step guide on how to create a PALM application. Note the fact that the PALM programs must include all of these steps but not necessarily in the given order.

1. Create a project file using the Code Warrior.
2. Set all the proper project setting in order to compile the project successfully.
3. Design the User Interface of the PALM program using the Constructor software. This is a program that comes with the Code Warrior Installer.
4. Save the interface files of the constructor programs into the "Src" folder in the Code Warrior project file.
5. Close the constructor and re-launch the code warrior project.
6. Add the constructor files to the project.
7. Include the ".h" files from the constructor software into the Code Warrior ".cpp" file in order to be able to link the User Interface objects and the program coding.
8. Program the application based on the standard application writing format for PALM devices. Refer to section 4.1.2 for an explanation of the basic programming loop used.
9. Download the program into the PALM emulator software.

[1] Refer to the previous section for a detailed explanation of each software that is mentioned.

10. Launch the debugger from the Code Warrior program in order to debug the application on the emulator.
11. Single step through the program and effectively debug the application.
12. Finalize the program and build a release version of it.
13. Download the program into the PALM device using the software that comes with the PALM (PALM Quick Install)

E.2.1.2 Debugging

The method of debugging a PALM application is rather unique as they are two methods of doing so. The first method is to download the program into the actual PALM device and debug it in the hardware itself. The downloading of the software is not done normally via the PALM Quick Install software which is provided upon purchasing the PALM. Instead, it is done by the programming environment where the program is not fully transferred to the PALM but instead swamped in and out in order to monitor the variables and changes done to it. The instructions that are swapped in are based on the single stepping that is done or breakpoints that are set in the program. In order to setup the PALM device to debug the program, it must be firstly changed to the debug mode. This is done by typing "ℓ..2" on the console with the stylus. In this mode, the PALM device is basically set to listen to whatever instruction that comes through the serial port. Nevertheless, manipulating the actual hardware is always never a good idea as corrupt programs might cause damage to the existing operating system on the PALM device. Hence, this is where the uniqueness and specialty of the second method comes into play

The second method is implemented by debugging the program using an external software called the PALM emulator. The PALM emulator is a software that is designed to simulate the exact workings of a PALM device on the PC environment. PALM applications can be loaded into the emulator and ran in order to see the exact response of the PALM device towards the application. Every emulator program needs a .ROM file in order to function as the PALM applications will be run in this file. A .ROM file is the content of a PALM device OS along with all the data and settings that are in it. As stated in the previous section, the two method of obtaining the .ROM file is either by downloading it from the PALM device or by applying for it online. As far as debugging

is concerned, the emulator can be linked to the Code Warrior software through certain setting. When the emulator is selected as the debugging tool and the debugger in Code Warrior is launched with the emulator running, the target application is automatically loaded into the emulator and the code is seen at its starting point. As illustrated with the actual hardware implementation, the developer can now single step through the program or set breakpoints in order to view that changes in the variables. The only difference between this method and the previous method is that the PALM device in this method is being simulated by the emulator.

E.2.2 Basic PALM Application Loop

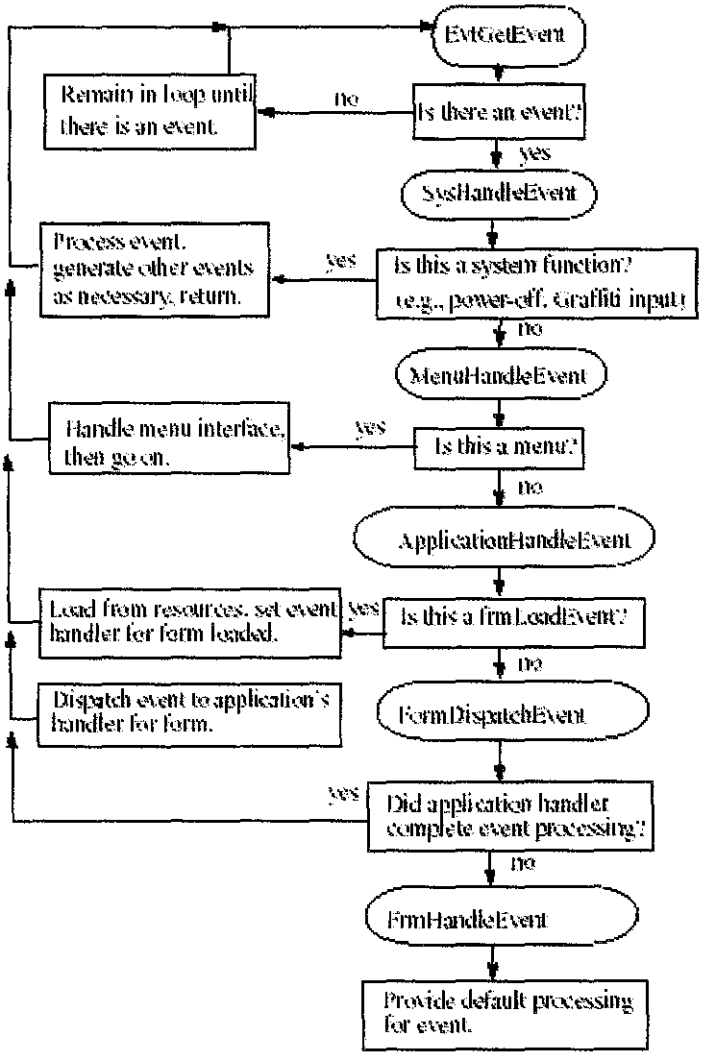
A PALM program is actually an infinite loop that is executed until an indication to exit the program is submitted. It is categorized as an event based program that takes into consideration every possible occurrence. Basically when an application is launched it goes into a loop that detects events. When an event is detected, the program sends the type of event into several handlers that would either process or ignore the event. Once all handlers have responded to the event, the program goes back to waiting for the next event. Hence, the definition an "event based" application. For everything that is done on a PALM device, an event is generated. For instance, if a stylus is placed on the PALM, and event "penDownEvent" is generated an sent into the program. This event contains a complete structure where it also provides the exact co-ordinates of the stylus, the resource that it is touching and etc. All in all, everything that happens is reported to the event loop and processed immediately. The event loop is **only** terminated if a terminate command is sent to the program.

In the event loop, the functions that will be called have been predetermined. There are several function that will be called by default in order to process each and every event. The following are the functions:-

1. PreprocessEvent
2. SystemHandleEvent
3. MenuHandleEvent

4. ApplicationHandleEvent

5. FrmDispatchEvent



The following is the typical source code for the event loop:-

```
static void EventLoop (void)
{
        Word error;
        EventType event;
        do
        {
                EvtGetEvent (&event, evtWaitForever);
                PreprocessEvent (&event);
                if (! SysHandleEvent (&event))
                if (! MenuHandleEvent (NULL, &event, &error))
```

```
        if (! ApplicationHandleEvent (&event))
        FrmDispatchEvent (&event);
    }
    while (event.eType != appStopEvent); // terminate program command received ?
}
```

The following is a step by step explanation of the given source code

1. Fetch an event from the event queue.
2. Call PreprocessEvent to allow the datebook event handler to see the command keys before any other event handler gets them. Some of the datebook views display UI that disappears automatically; this UI needs to be dismissed before the system event handler or the menu event handler displays any UI objects. Note that not all applications need a PreprocessEvent function. It may be appropriate to call SysHandleEvent right away.
3. Call SysHandleEvent to give the system an opportunity to handle the event. The system handles events like power on/power off, Graffiti input, tapping silk-screened icons, or pressing buttons. During the call to SysHandleEvent, the user may also be informed about low-battery warnings or may find and search another application. Note that in the process of handling an event, SysHandleEvent may generate new events and put them on the queue. For example, the system handles Graffiti input by translating the pen events to key events. Those, in turn, are put on the event queue and are eventually handled by the application. SysHandleEvent returns true if the event was completely handled, that is, no further processing of the event is required. The application can then pick up the next event from the queue.
4. If SysHandleEvent did not completely handle the event, the application calls MenuHandleEvent. MenuHandleEvent handles two types of events:
    - If the user has tapped in the area that invokes a menu, MenuHandleEvent brings up the menu.
    - If the user has tapped inside a menu to invoke a menu command, MenuHandleEvent removes the menu from the screen and puts the

events that result from the command onto the event queue. MenuHandleEvent returns TRUE if the event was completely handled.

5. If MenuHandleEvent did not completely handle the event, the application calls ApplicationHandleEvent, a function the application has to provide itself. ApplicationHandleEvent handles only the frmLoadEvent for that event; it loads and activates application form resources and sets the event handler for the active form.
6. If ApplicationHandleEvent did not completely handle the event, the application calls FrmDispatchEvent. FrmDispatchEvent first sends the event to the application's event handler for the active form. This is the event handler routine that was established in ApplicationHandleEvent. Thus the application's code is given the first opportunity to process events that pertain to the current form. The application's event handler may completely handle the event and return true to calls from FrmDispatchEvent.In that case, FrmDispatchEvent returns to the application's event loop. Otherwise, FrmDispatchEvent calls FrmHandleEvent to provide the system's default processing for the event. For example, in the process of handling an event, an application frequently has to first close the current form and then open another one, as follows:
   - The application calls FrmGotoForm to bring up another form. FrmGotoForm queues a frmCloseEvent for the currently active form, then queues frmLoadEvent and frmOpenEvent for the new form.
   - When the application gets the frmCloseEvent, it closes and erases the currently active form.
   - When the application gets the frmLoadEvent, it loads and then activates the new form. Normally, the form remains active until it's closed. The application's event handler for the new form is also established.
   - When the application gets the frmOpenEvent, it performs any required initialization of the form, and then draws the form on the display.
7. After FrmGotoForm has been called, any further events that come through the main event loop and to FrmDispatchEvent are dispatched to the event handler for the form that's currently active. For each dialog box or form, the event handler

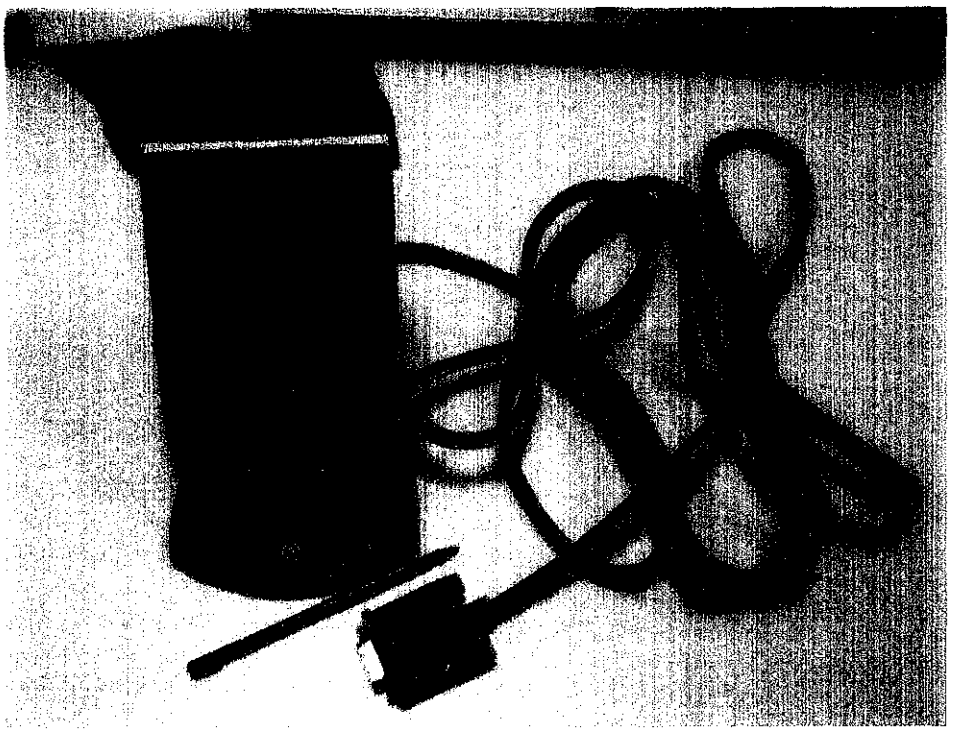
knows how it should respond to events, for example, it may open, close, highlight, or perform other actions in response to the event. FrmHandleEvent invokes this default UI functionality. After the system has done all it can to handle the event for the specified form, the application finally calls the active form's own event handling function.

Appendix F

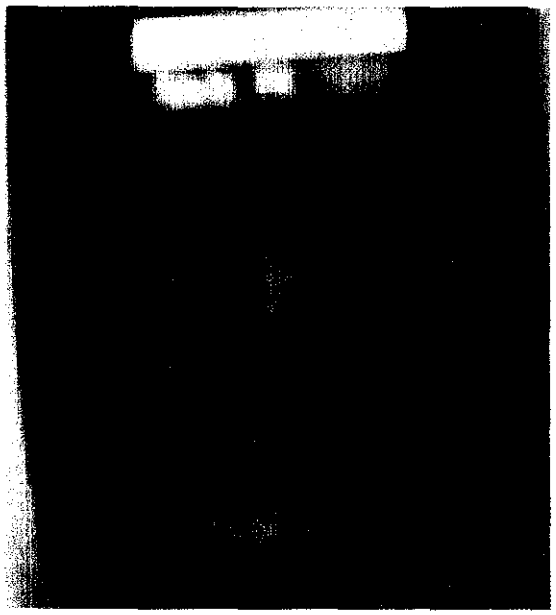
PALM Hardware Images

APPENDIX F: PALM Hardware Images

This is the PALM IIIxe cradle. The Palm will be placed on the cradle in order to connect to the serial port. The flat contacts seen on the cradle is connected directly to the pins in the female DB-9 connector



The Palm placed on the cradle. The serial port on the Palm (refer to the following image) connect directly to the flat contacts on the cradle.

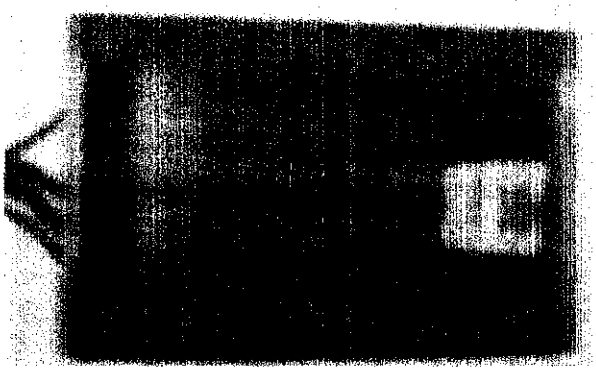


The Palm serial port

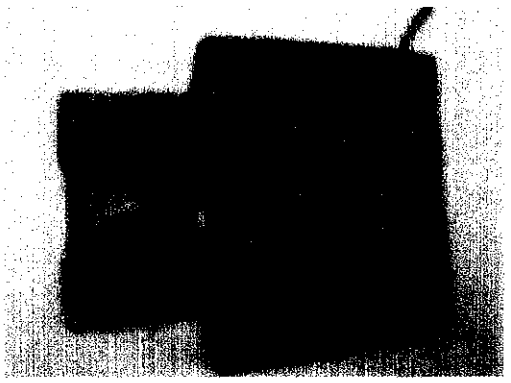
Appendix G

Universal "Chip Based" Smart Card Snooping Device Snap Shots

APPENDIX G: Universal "Chip Based" Smart Card Snooping Device Snap Shots

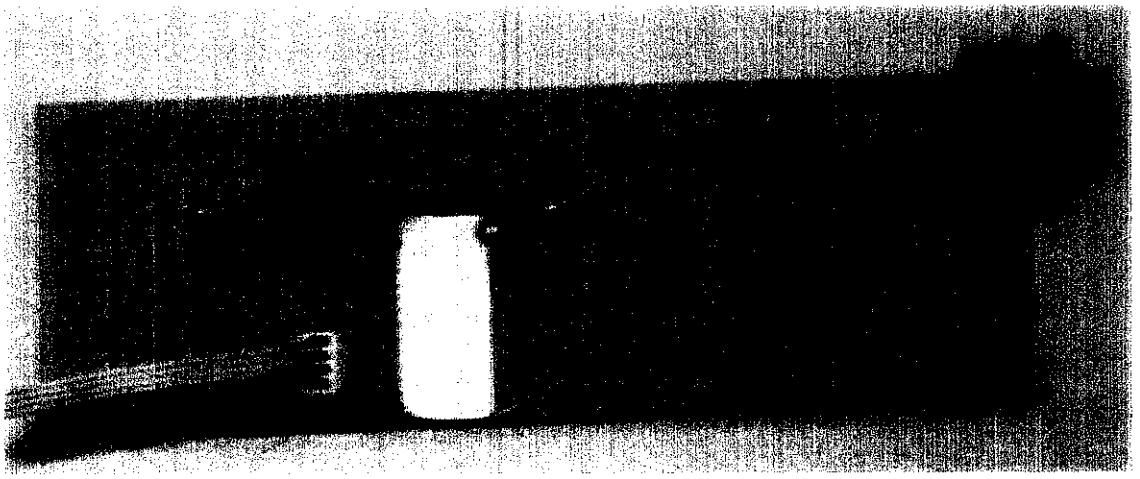


Dummy Smart Card

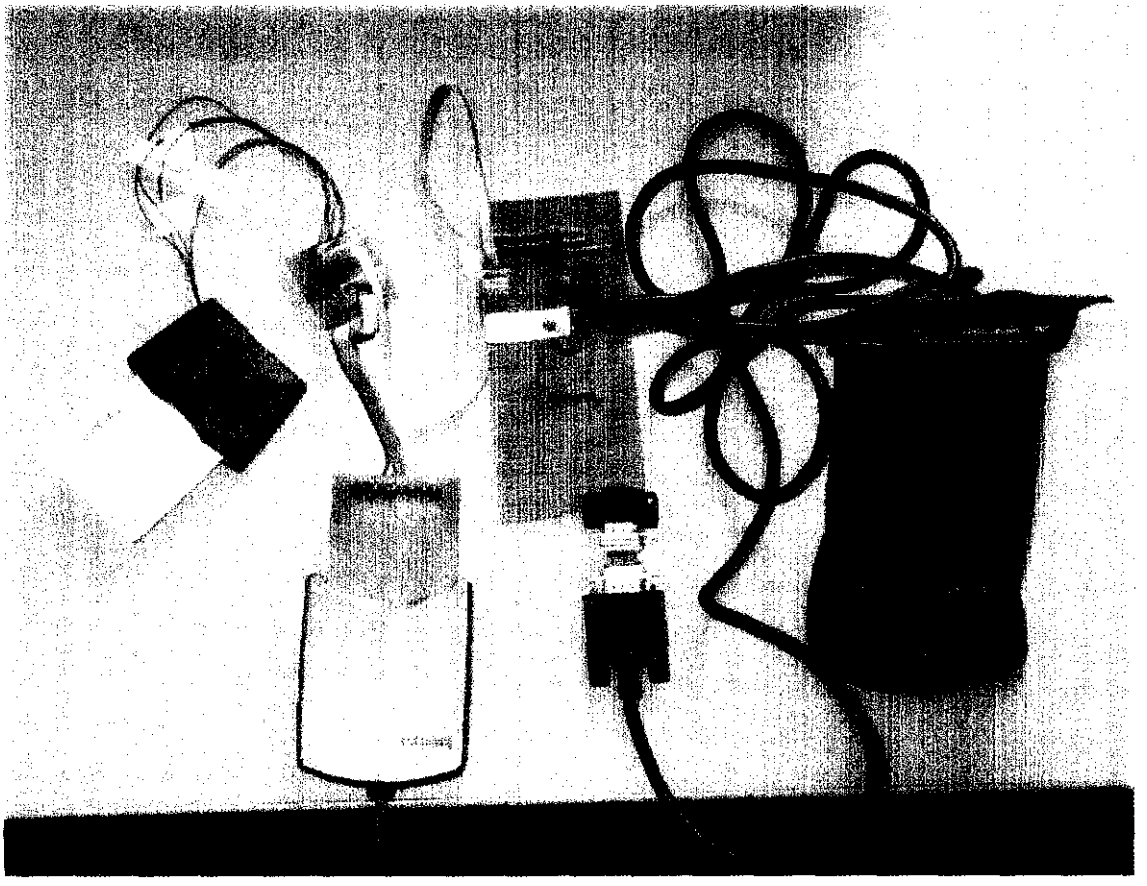


Connector

Snooping module



Communication Module



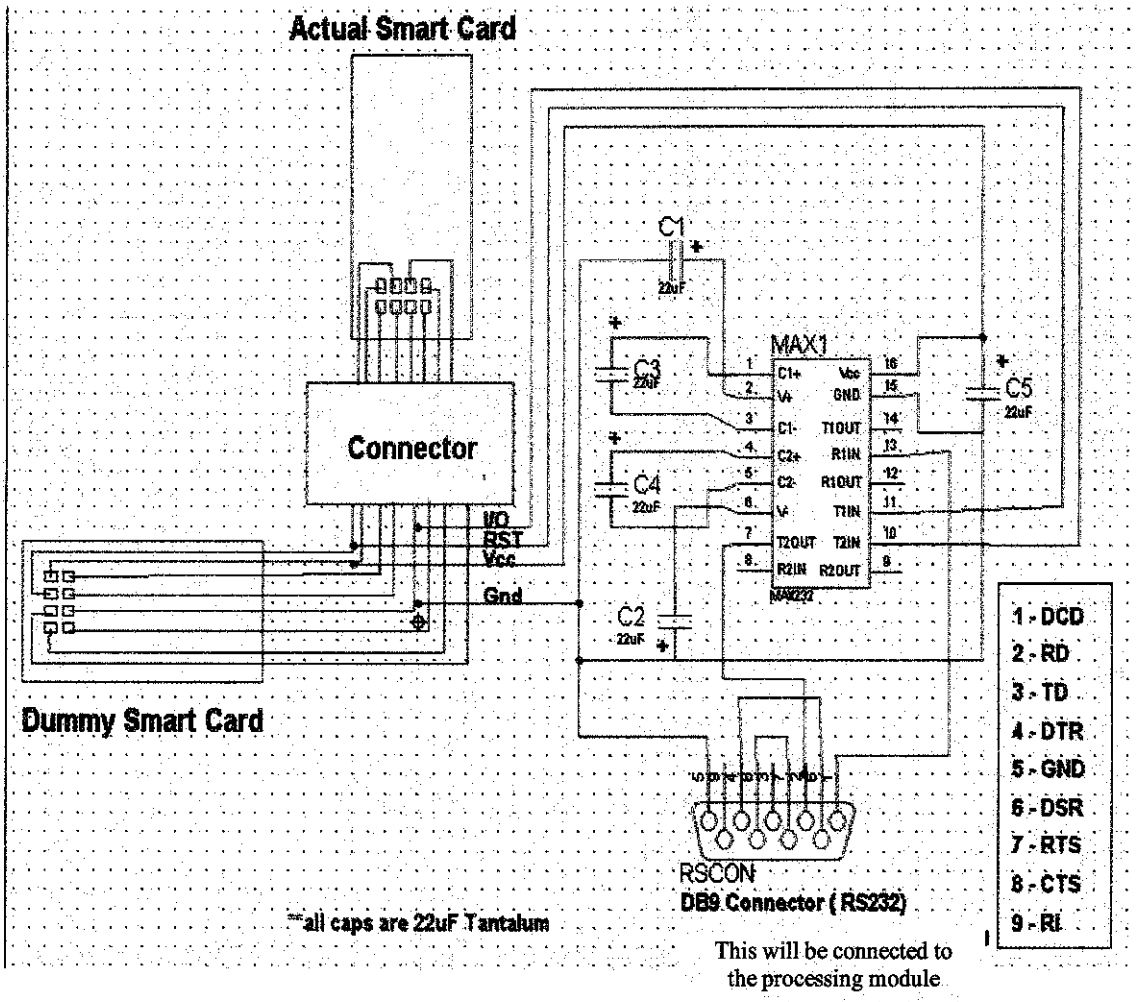
The complete hardware setup for the snooping device

The figure above illustrates the complete setup for all three modules and the method as to which it is connected. The white card is the actual smart card that is snooped on and the gray reader is the Schlumberger Reflex 72 v2 smart card reader.

Appendix H

Detailed Schematics of the Snooping and Communication Module

APPENDIX H: Detailed Schematics of the Snooping and Communication Module



Appendix I

ASCII Character Codes

APPENDIX I: ASCII Character Codes

Ctrl	Dec	Hex	Char	Code
^@	0	00		NUL
^A	1	01	☺	SOH
^B	2	02	☻	STX
^C	3	03	♥	ETX
^D	4	04	♦	EOT
^E	5	05	♣	ENQ
^F	6	06	♠	ACK
^G	7	07	•	BEL
^H	8	08	◘	BS
^I	9	09	○	HT
^J	10	0A	◙	LF
^K	11	0B	♂	VT
^L	12	0C	♀	FF
^M	13	0D	♪	CR
^N	14	0E	♫	SO
^O	15	0F	☼	SI
^P	16	10	►	DLE
^Q	17	11	◄	DC1
^R	18	12	↕	DC2
^S	19	13	‼	DC3
^T	20	14	¶	DC4
^U	21	15	§	NAK
^V	22	16	▬	SYN
^W	23	17	↨	ETB
^X	24	18	↑	CAN
^Y	25	19	↓	EM
^Z	26	1A	→	SUB
^[	27	1B	←	ESC
^\	28	1C	∟	FS
^]	29	1D	↔	GS
^^	30	1E	▲	RS
^_	31	1F	▼	US

Dec	Hex	Char
32	20	SP
33	21	!
34	22	"
35	23	#
36	24	$
37	25	%
38	26	&
39	27	'
40	28	(
41	29	)
42	2A	*
43	2B	+
44	2C	,
45	2D	-
46	2E	.
47	2F	/
48	30	0
49	31	1
50	32	2
51	33	3
52	34	4
53	35	5
54	36	6
55	37	7
56	38	8
57	39	9
58	3A	:
59	3B	;
60	3C	<
61	3D	=
62	3E	>
63	3F	?

Dec	Hex	Char
64	40	@
65	41	A
66	42	B
67	43	C
68	44	D
69	45	E
70	46	F
71	47	G
72	48	H
73	49	I
74	4A	J
75	4B	K
76	4C	L
77	4D	M
78	4E	N
79	4F	O
80	50	P
81	51	Q
82	52	R
83	53	S
84	54	T
85	55	U
86	56	V
87	57	W
88	58	X
89	59	Y
90	5A	Z
91	5B	[
92	5C	\
93	5D	]
94	5E	^
95	5F	_

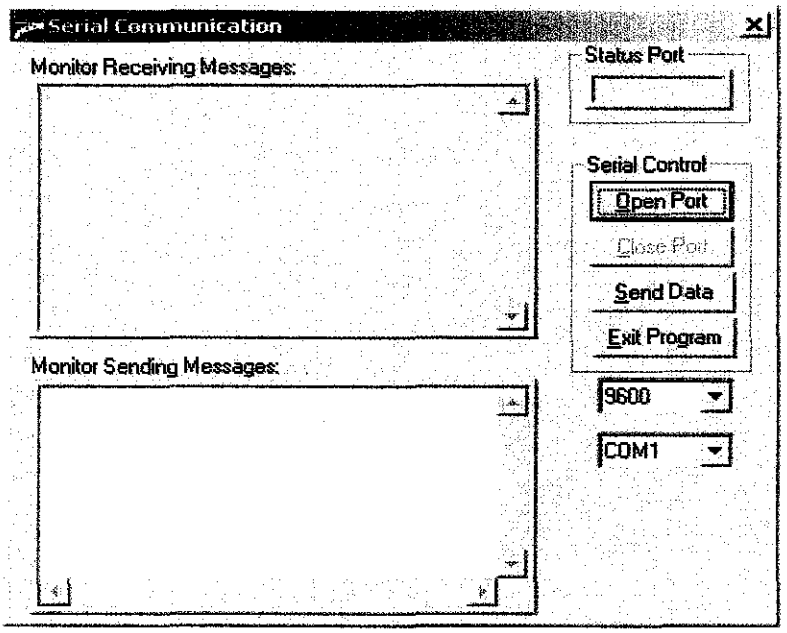
Dec	Hex	Char
96	60	`
97	61	a
98	62	b
99	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o
112	70	p
113	71	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7A	z
123	7B	{
124	7C	|
125	7D	}
126	7E	~
127	7F	⌂†

Appendix J

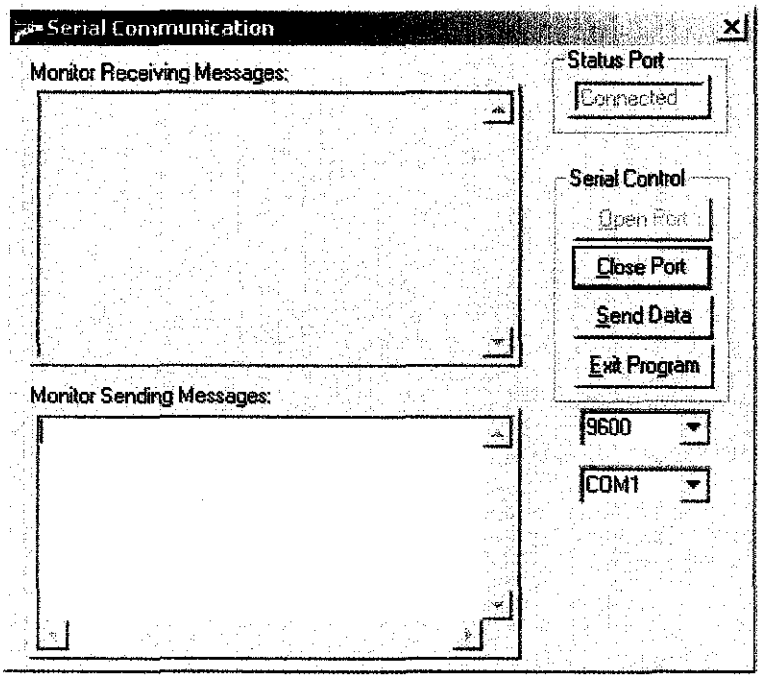
Screenshots of the Visual C++ program

APPENDIX J: Screenshots of the Visual C++ Program

This is the initial user interface when the program is started

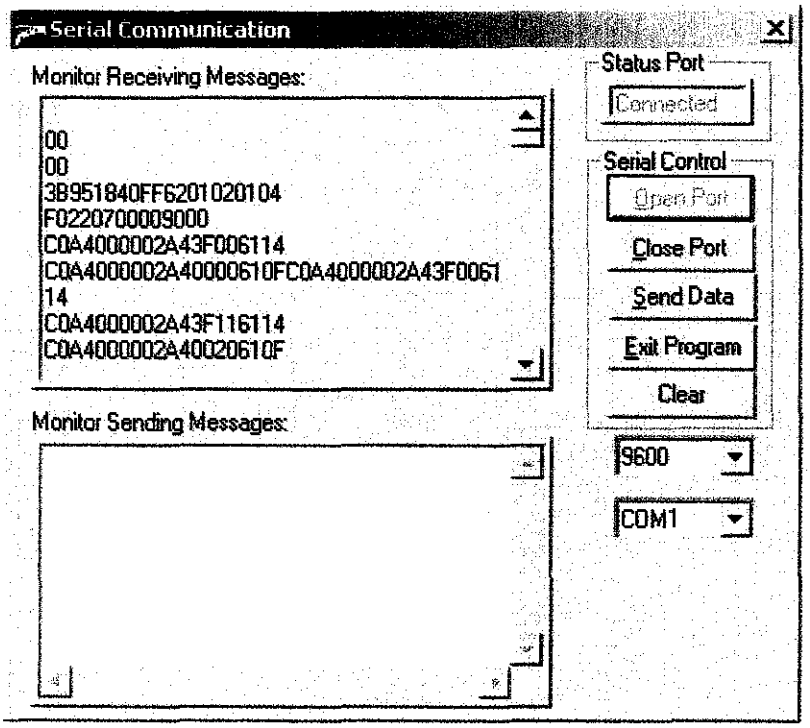


After selecting the serial port and the Baud rate, the "Open Port" button is pressed



We are now connected to the serial port

The RS232 (DB-9) male connector of the the PC is connected to the female connector on the communication module. Hence, all data transfer between the card and the reader would be logged.



The receive pin picks up the data (as it is connected to the communication module) and displays it in the received text box.

Appendix K

PALM OS Functions and Structures

APPENDIX K: PALM OS C Functions and Structures

The following are the descriptions of some of the most important functions that were used in order to establish serial communication. In addition to that, the structure that registers the setting for the serial port is also illustrated at the end.

K.1 SysLibFind function

Purpose Return a reference number for a library that is already loaded, given its name.

Prototype `Err SysLibFind (CharPtr nameP, UIntPtr refNumP)`

Parameters

`nameP` Pointer to the name of a loaded library.

`refNumP` Pointer to a variable for returning the library reference number (on failure, this variable is undefined)

Result 0 if no error; otherwise: `sysErrLibNotFound` (if the library is not yet loaded), or another error returned from the library's install entry point.

Comments Most built-in libraries (NetLib, serial, IR) are preloaded automatically when the system is reset. Third-party libraries must be loaded before this call can succeed (use `SysLibLoad`). You can check if a library is already loaded by calling `SysLibFind` and checking for a 0 error return value (it will return a non-zero value if the library is not loaded).

K.2 SerOpen function

Purpose Acquire and open a serial port with given baud rate and default settings.

Prototype `Err SerOpen (UInt refNum, UInt port, ULong baud)`

Parameters

`refNum` Serial library reference number.

`port` Port number.

`baud` Baud rate.

Result 0 No error.

`serErrAlreadyOpen` Port was open. Enables port sharing by "friendly" clients (not recommended).

`serErrBadParam` Invalid parameter.

`memErrNotEnoughSpace` Insufficient memory.

Comments Acquires the serial port, powers it up, and prepares it for operation.To obtain the serial library reference number, call SysLibFind with "Serial

Library" as the library name. This reference number must be passed as a parameter to all serial manager functions. The device currently contains only one serial port with port number 0 (zero).
The baud rate is an integral baud value (for example - 300, 1200, 2400, 9600, 19200, 38400, 57600, etc.). The Palm OS device has been tested at the standard baud rates in the range of 300 - 57600 baud. Baud rates through 1 Mbit are theoretically possible. Use CTS and shaking at baud rates above 19200 (see `SerSetSettings`). An error code of 0 (zero) or `serErrAlreadyOpen` indicates that the port was successfully opened. If the port is already open when `SerOpen` is called, the port's open count is incremented and an error code of `serErrAlreadyOpen` is returned. This ability to open the serial port multiple times allows cooperating tasks to share the serial port. Other tasks must refrain from using the port if `serErrAlreadyOpen` is returned and close it by calling `SerClose`.

K.3 SerSetSetting function

Purpose Set the serial port settings; that is, change its attributes.

Prototype
```
Err SerSetSettings (UInt refNum, SerSettingsPtr settingsP)
```

Parameters

`refNum` Serial library reference number.

`settingsP` Pointer to the filled in `SerSettingsType` structure.

Result 0 No error.
`serErrNotOpen` The port wasn't open.
`serErrBadParam` Invalid parameter.

Comments The attributes set by this call include the current baud rate, CTS timeout,handshaking options, and data format options. See the definition of the `SerSettingsType` structure for more details.

K.4 SerReceiveCheck function

Purpose Return the count of bytes presently in the receive queue.

Prototype `Err SerReceiveCheck (UInt refNum, ULongPtr numBytesP)`

Parameters

`refNum` Serial library reference number.

`numBytesP` Pointer to location for returning the byte count.

Result 0 No error.

`serErrLineErr` Line error pending (see `SerClearErr` and `SerGetStatus`).

Comments Because this call does not return the byte count if line errors are pending, it is important to acknowledge the detection of line errors by calling `SerClearErr`.

K.5 SerReceiveFlush function

Purpose Discard all data presently in the receive queue and flush bytes coming into the serial port. Clear the saved error status.

Prototype `void SerReceiveFlush (UInt refNum, Long timeout)`

Parameters

`refNum` Serial library reference number.

`timeout` Interbyte time out in system ticks (-1 = forever).

Result Returns nothing.

Comments `SerReceiveFlush` blocks until a timeout occurs while waiting for the next byte to arrive.

K.6 SerReceive function

Purpose Receives `size` bytes worth of data or returns with error if a line error or timeout is encountered.

Prototype `ULong SerReceive (UInt refNum, VoidPtr rcvBufP, ULong count, Long timeout, Err* errP)`

Parameters

`refNum` Serial library reference number.
`rcvBufP` Buffer for receiving data.
`count` Number of bytes to receive.
`timeout` Interbyte timeout in ticks, 0 for none, -1 forever.
`errP` For returning error code.

Result Number of bytes received:
`*errP = 0` No error.
`serErrLineErr` RS232 line error.
`serErrTimeOut` Interbyte timeout.

K.7 SerSettingType function

The `SerSettingsType` structure defines serial port attributes; it is used by the calls SerGetSettings and SerSetSettings. The `SerSettingsPtr` type points to a `SerSettingsType` structure.

```
typedef struct SerSettingsType
{
UInt32 baudRate;
UInt32 flags;
Long ctsTimeout;
}
SerSettingsType;
typedef SerSettingsType* SerSettingsPtr;
```

Field Descriptions

`baudRate` Baud rate
`flags` Miscellaneous settings
`ctsTimeout` Maximum number of ticks to wait for CTS to become asserted before transmitting; used only when configured with the `serSettingsFlagCTSAutoM` flag.

Appendix L

Answer To Reset (ATR)

APPENDIX L: ANSWER TO RESET (ATR)

Character ID	Definition
	Initial Character Section
TS	Mandatory initial character
	Format Character Section
T0	Indicator for presence of interface characters
	Interface Character Section
TA_1	Global, codes F1 and D1
TB_1	Global, codes 11 and Pl1
TC_1	Global, code N
TD_1	Codes Y_2 and T
TA_2	Specific
TB_2	Global, code Pl2
TC_2	Specific
TD_2	Codes Y_3 and T
TA_3	TA_i, TB_i and TC_i are specific
...TD_i	Codes Y_{i+1} and T
	Historical Character Section
T1	Card specific information
...TK	(Maximum of 15 characters)
	Check Character Section
TCK_2	Optional check character

For a even more detailed specification and explanation of the ATR , refer to http://www.hackersrussia.ru/Cards/Syncro/ISO7816.php

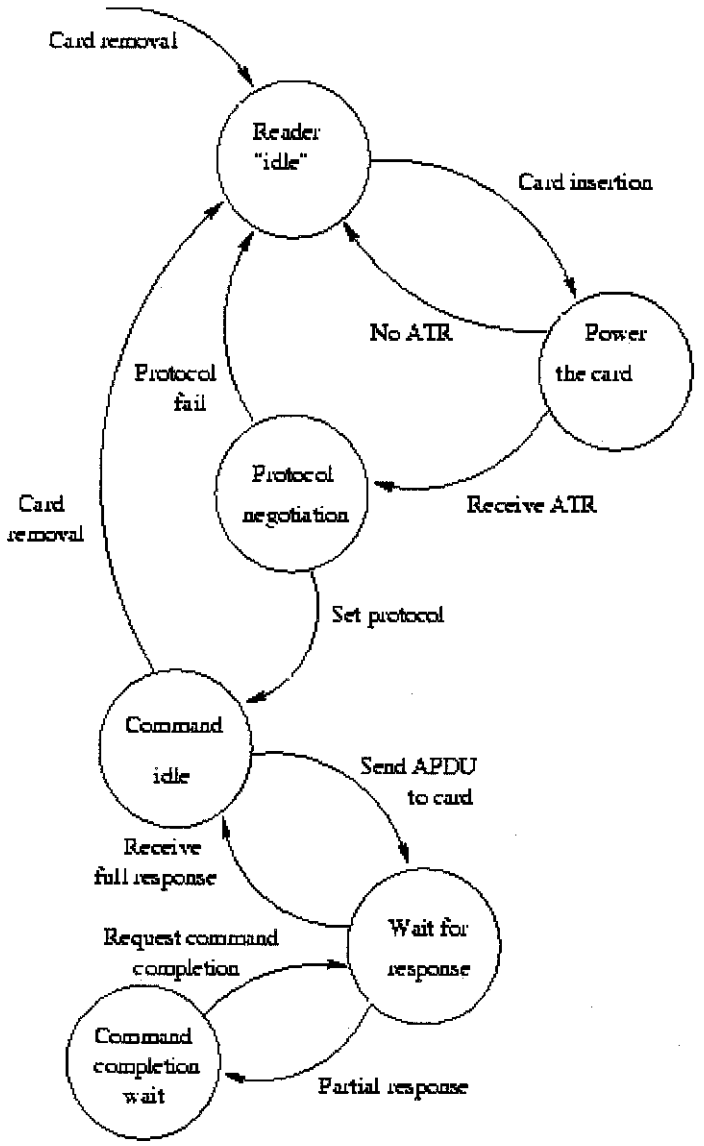
Appendix M

Smart Card Communication State Diagram

APPENDIX M: Smart Card Communication State Diagrams

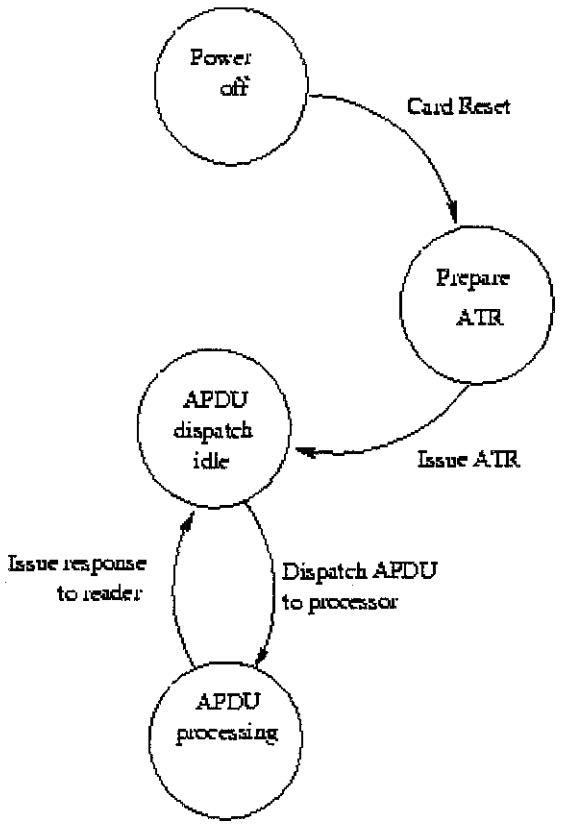
Communication between smart card and reader through a series of state transitions

Appendix M .1



Reader State Diagram

Appendix M .2



Card State Diagram

Appendix N

Complete Source Code for the PALM Snooping Program

```
*****************************************************************\
                                                                 ||
                                                                 ||
Title:      Universal "Chip Based Smart Card" Snooping Program   ||
            ||                                                   ||
Version:    1.00                                                 ||
File Name:  Snooper.c                                            ||
Author:     Dimitri Denamany                                     ||
Date:       March 27, 2004                                       ||
Language:   C                                                    ||
                                                                 ||
Purpose:    To snoop  on the data communication lines between the ||
            smart card and the smart card reader                 ||
                                                                 ||
                                                                 ||
*****************************************************************/


clude <Pilot.h>
clude <SysEvtMgr.h>
clude <SerialMgr.h>
clude <Field.h>
clude <Window.h>
clude "Snooper_res.h"

*****************************************************************\
  Internal Constants                                             ||
*****************************************************************/



fine appFileCreator           'BBPT'
fine appVersionNum            0x01
fine appPrefID                0x00
fine appPrefVersionNum        0x01
fine TRANSMIT_STRING_LENGTH   24
fine MAX_LENGTH               60000
fine HELP_TEXT_LENGTH         255
fine PORT                     0
fine errNone                  0


lerts display only if in Debug Mode which is defined in stdhdr.h
se: dbgAlert(DebugAlert, "String1", "String2", "")
his will display a message box with "String1: String2"
  (debug)
fine dbgAlert(a, b, c, d) FrmCustomAlert(a, b, c, d)
se
fine dbgAlert(a, b, c, d)
dif

Define the minimum OS version we support
fine ourMinVersion    sysMakeROMVersion(2,0,0,sysROMStageRelease,0)


*****************************************************************\
  Global variables                                               ||
*****************************************************************/

lean                      gSerialPortOpen;
t                         gSerRefNumber;
lean                      gIsHex;
d                         gPrefsSize = 0;
r                         gReceivedFromExternal;
r                         gTransmittedFromPalm;
r                         gMessageToDisplay[TRANSMIT_STRING_LENGTH];
ng                        Baud ;

*****************************************************************\
Internal Function Prototypes                                     ||
*****************************************************************/

d ClearScreenData(Word objectID);
d CloseSerialPort(void);
d OpenSerialPort(void);
d ReceiveData(void);
d ConvertMessageToDisplay();
d DisplayData(FieldType* FieldP);


d SaveTransmitStrings();
d LoadTransmitStrings();
d TransmitString(char strBuffer[TRANSMIT_STRING_LENGTH]);
d RetrieveTransmitStrings(Word TextToTransmit);
```

```
it IsValidHex(char* chrBuffer);
it ConvertToDec(char* chrBuffer);
d ConvertToHex(Int NumToConvert);

d MainFormInit(FormPtr frmP);
lean MainFormDoCommand(Word command);
lean MainFormHandleEvent(EventPtr eventP);

d TextFormInit(FormPtr FrmP);
lean TextFormHandleEvent(EventPtr eventP);

dPtr GetObjectPtr(Word objectID);
 RomVersionCompatible(DWord requiredVersion, Word launchFlags);
lean AppHandleEvent( EventPtr eventP);
d AppEventLoop(void);
 AppStart(void);
d AppStop(void);
rd SerialCommPilotMain(Word cmd, Ptr cmdPBP, Word launchFlags);
rd PilotMain( Word cmd, Ptr cmdPBP, Word launchFlags);

tic void Scroll ( int flag ) ;
tic void UpdateScrollBar () ;
tic void EditViewScroll (Short linesToScroll);


*************************************************************************\
   Internal Functions                                                    ||
*************************************************************************/


*************************************************************************\
                                                                         ||
 FUNCTION:    ClearScreenData                                            ||
                                                                         ||
 DESCRIPTION: Clear data from Transmit and Receive Fields                ||
                                                                         ||
 PARAMETERS:  FieldToClear - Which field to clear                        ||
                                                                         ||
 RETURNED:    Nothing                                                    ||
                                                                         ||
 REVISION HISTORY:                                                       ||
                                                                         ||
*************************************************************************/
d ClearScreenData(Word FieldToClear)

  FieldType*  CurrentField;
  Handle      TextHandle;

  CurrentField = (FieldPtr)(GetObjectPtr(FieldToClear));          //Get pointer to field
  TextHandle = FldGetTextHandle(CurrentField);                    //Get memory handle associate
ith field

  if (TextHandle)                                                 //Will return NULL if no text
      {
      FldSetTextHandle(CurrentField, NULL);                       //Set handle of the current f
d to NULL
      MemHandleFree(TextHandle);                                  //Free memory associated with
e handle
      FldDrawField(CurrentField);                                 //Refresh field
      }
  //End Function ClearScreenData


*************************************************************************\
                                                                         ||
 FUNCTION:    CloseSerialPort                                            ||
                                                                         ||
 DESCRIPTION: Close serial port                                          ||
                                                                         ||
 PARAMETERS:  none                                                       ||
                                                                         ||
 RETURNED:    nothing                                                    ||
                                                                         ||
 REVISION HISTORY:                                                       ||
                                                                         ||
*************************************************************************/
d CloseSerialPort(void)

  Err         error = 0;
  FormPtr     frmP;
  Word        ObjectIndex;
```

```
  error = SerSendWait(gSerRefNumber, -1);                          //Make sure all data has been
nt before exiting
  if (error == serErrTimeOut) FrmAlert(PortTimeoutAlert);          //If SerSendWait times out th
display timeout alert
  SerClose(gSerRefNumber);                                         //Close Serial Port
  frmP = FrmGetActiveForm();                                       //Get Pointer to Active Form

  //The following two lines hide the port open icon.
  //If you are connected to the serial port,
  //the icon is displayed, otherwise there is nothing there.
  ObjectIndex = FrmGetObjectIndex(frmP, Main_PortOpenBitMap);
  FrmHideObject(frmP, ObjectIndex);

  gSerialPortOpen = false;                                         //Set PortOpen to False

  //End Function CloseSerialPort


*****************************************************************\
                                                                  ||
 FUNCTION:    OpenSerialPort                                      ||
                                                                  ||
 DESCRIPTION: Open serial port                                    ||
                                                                  ||
 PARAMETERS:  none                                                ||
                                                                  ||
 RETURNED:    nothing                                             ||
                                                                  ||
 REVISION HISTORY:                                                ||
                                                                  ||
*****************************************************************/
d OpenSerialPort(void)

  Err             error = 0;
  DWord           CommFlags = 0;
  Word            SizeOfFlags = sizeof(CommFlags);
  SerSettingsType SerCommSettings;
  FormPtr         frmP;
  Word            ObjectIndex;

  error = SysLibFind("Serial Library", &gSerRefNumber);            //Get Serial Reference Number

  if (error)
      {
      FrmAlert(OpenPortAlert);                                     //Default Open Serial Port fa
d error message
      }

  error = SerOpen(gSerRefNumber, PORT, Baud);                      //Open Serial Port 9600 baud,
rtID returned to gSerialPortID

  switch (error)                                                   //Error checking
  {
    case serErrAlreadyOpen:                                        //Serial Port is already open
 another application
      SerClose(gSerRefNumber);                                     //Close port: problems can ar
 by sharing the serial port,
      FrmAlert(PortBusyAlert);                                     //Display port busy message
      break;
    case errNone:                                                  //No errors returned, port op
d successfully
      gSerialPortOpen = true;                                      //Set flag to port open
      frmP = FrmGetActiveForm();                                   //Get Pointer to Active Form
      //The following two lines show the port open icon.
      //If you are connected to the serial port,
      //the icon is displayed, otherwise there is nothing there.
      ObjectIndex = FrmGetObjectIndex(frmP, Main_PortOpenBitMap);
      FrmShowObject(frmP, ObjectIndex);
      break;
    default:                                                       //Other Error
      FrmAlert(OpenPortAlert);                                     //Default Open Serial Port fa
d error message
      break;
  }

  SerReceiveFlush(gSerRefNumber, 200);                             //Clear port in case last app
ation didn't


  //Define Communications settings, set to default settings, 8,N,1, No Flow Control,
  //Since this is the default it can be omitted
  CommFlags = serSettingsFlagBitsPerChar8 | serSettingsFlagStopBits2 | serSettingsFlagParityEve
```

```
  SerCommSettings.baudRate = Baud;                                    //Set baud rate
  SerCommSettings.flags = CommFlags;                                  //Set flags

  error = SerSetSettings(gSerRefNumber, &SerCommSettings);            //Set communication settings

  if (error) FrmAlert(CommSettingsAlert);                             //Set Comm Settings failed er
r message



  //End Function OpenSerialPort



*****************************************************************************\
                                                                              ||
 FUNCTION:    ReceiveData                                                     ||
                                                                              ||
 DESCRIPTION: Receive data through serial port                                ||
                                                                              ||
 PARAMETERS:  none                                                            ||
                                                                              ||
 RETURNED:    Nothing                                                         ||
                                                                              ||
 REVISION HISTORY:                                                            ||
                                                                              ||
*****************************************************************************/
id ReceiveData(void)

  Err         error = 0;
  DWord       NumberOfBytes;
  char        chrArray;
  Word        i = 0;
  FieldType   *FieldP;
  char        tmpBuffer;
  //VoidPtr  pBuffer;
  //pBuffer = new char [9] ;

  if (!(gSerialPortOpen)) return;                                     //Not connected

  error = SerReceiveCheck(gSerRefNumber, &NumberOfBytes);             //Retrieve the number of byte
:o be retrieved

  if (error == serErrLineErr) SerClearErr(gSerRefNumber);             //Clear Error on Line Error
  if (error)                                                          //If error, notify user unabl
:o retrieve data.
      {
      //FrmAlert(CheckPortAlert);                                     //Receive failed error messag

      return;
      }

  if ((NumberOfBytes + i) > MAX_LENGTH)                               //If there are more bytes wai
ıg than there is buffer space
    {
    NumberOfBytes = MAX_LENGTH - i - 1;                               //MaxBuffer - Array Index - 1
ır terminating null
    }



  //    pBuffer = new char[NumberOfBytes] ;
  while (NumberOfBytes)                                               //Retrieve data
      {
      SerReceive(gSerRefNumber, &chrArray, 1, 0, &error);             //Retrieve byte one at a time
to chrArray

      if (error == serErrLineErr) SerClearErr(gSerRefNumber);         //Clear Error on Line Error

      if (error)
          {
          SerReceiveFlush(gSerRefNumber, 1);                          //Clear buffer on error
          i = 0;                                                      //Reset Array Index
          NumberOfBytes = 0;
          return;
          }
      else
          {
          i++;                                                        //Increase Array Index
          NumberOfBytes--;                                            //Decrease number of bytes to
  retrieved
          FieldP = GetObjectPtr(Main_ReceivedFromExternalField);      //Get Pointer to Field
          StrCopy(&gMessageToDisplay[0], &chrArray);                  //Copy into display string
```

```
          StrCopy(&gMessageToDisplay[1], "");                    //Terminate with NULL
          ConvertMessageToDisplay();                             //Convert character to hex if
n-printable
          DisplayData(FieldP);                                   //Display Data into above Fie


          UpdateScrollBar () ;


          chrArray = '/0';                                       //Reset array
          tmpBuffer = '/0';                                      //Reset buffer
          }
      }   // end while
      //UpdateScrollBar () ;
  //End Function ReceiveData


*****************************************************************\
                                                                 ||
 FUNCTION:    ConvertMessageToDisplay                            ||
                                                                 ||
 DESCRIPTION: This routine displays the data to hex format       ||
              if necessary                                       ||
                                                                 ||
 PARAMETERS:  none                                               ||
                                                                 ||
 RETURNED:    nothing                                            ||
                                                                 ||
 REVISION HISTORY:                                               ||
                                                                 ||
*****************************************************************/
id ConvertMessageToDisplay()

  Int DecNumberToConvert;

  DecNumberToConvert = (int)((unsigned char)(gMessageToDisplay[0]));  //Get character

  ConvertToHex(DecNumberToConvert);                              //Convert to hex


*****************************************************************\
                                                                 ||
 FUNCTION:    DisplayData                                        ||
                                                                 ||
 DESCRIPTION: This routine display to a field                    ||
                                                                 ||
 PARAMETERS:  FieldP  - the field to display the message in      ||
                                                                 ||
 RETURNED:    NOTHING                                            ||
                                                                 ||
 REVISION HISTORY:                                               ||
                                                                 ||
*****************************************************************/
id DisplayData(FieldType* FieldP)

  Err          error = 0;
  Handle       TextHandle = 0;
  char         *tmpBuffer;
  char         *tmpText1;
  char         *tmpText2;
  Word         tmpBufferLength;
  Word         MessageToDisplayLength;
  Word         CharactersToRemove;
  Int          i = 0;

  MessageToDisplayLength = StrLen(gMessageToDisplay);                  //Get length of strin
to display

  TextHandle = FldGetTextHandle(FieldP);                               //Get handle of field
  if (TextHandle)                                                      //If there is text in
he field
      {
      FldSetTextHandle(FieldP, NULL);                                  //Strip Text

      tmpBuffer = MemHandleLock(TextHandle);                           //Retrieve the text o
 of the field
      tmpBufferLength = StrLen(tmpBuffer);                             //Get length of strin
in field
      if (tmpBufferLength < MAX_LENGTH)                                //Text field string i
less than maximum
          {
```

```
        MemHandleUnlock(TextHandle);                                    //Unlock handle, resi
to accommodate new data
        error = MemHandleResize(TextHandle, min(MAX_LENGTH + 1,         /*Plus one for NULL*/
                tmpBufferLength + MessageToDisplayLength + 1));         //Plus one for NULL
        if (error) FrmAlert(MemoryAlert);                               //Out of memory Alert
        tmpBuffer = MemHandleLock(TextHandle);                          //Lock the text field
etrieve text
        } //End if
    if (tmpBufferLength + MessageToDisplayLength > MAX_LENGTH)          //If final string is
 big
        {
        CharactersToRemove = MessageToDisplayLength + tmpBufferLength - //Determine number of
aracters to remove
                        MAX_LENGTH;
        tmpText2 = &tmpBuffer[CharactersToRemove];                      //Start tmpBuffer at
t character to remove
        for (tmpText1 = tmpBuffer; *tmpText2 != chrNull; ++tmpText1, ++tmpText2)
            {                                                           //While there are cha
ters to remove, increment array
            *tmpText1 = *tmpText2;                                      //Transfer character
Buffer
            } //End for
        *tmpText1 = chrNull;                                            //Terminate null
        } //Endif

        StrCat(tmpBuffer, gMessageToDisplay);                           //Add new message to
 of old message
    }
  else
    {                                                                   //If the field was em
r
    TextHandle = MemHandleNew(MessageToDisplayLength + 1);              //Create memory for i
rmation
    if (TextHandle == chrNull) FrmAlert(MemoryAlert);                   //Out of memory alert
    tmpBuffer = MemHandleLock(TextHandle);                              //Lock handle
    StrCopy(tmpBuffer, gMessageToDisplay);                              //Replace memory with
ssage

    }
 MemHandleUnlock(TextHandle);                                           //Unlock Handle
 FldSetTextHandle(FieldP, TextHandle);                                  //Set handle to field
 FldDrawField(FieldP);                                                  //Refresh field
 gMessageToDisplay[0] = chrNull;                                        //Reset message to di
ay


agma mark ----------------


*****************************************************************\
                                                                 | |
FUNCTION:    ConvertToHex                                        | |
                                                                 | |
DESCRIPTION: Convert Decimal Numbers to Hex, surrounded by <>    | |
                                                                 | |
PARAMETERS:  NumToConvert  - Decimal to convert to Hexadecimal   | |
                                                                 | |
RETURNED:    nothing                                             | |
                                                                 | |
REVISION HISTORY:                                                | |
                                                                 | |
                                                                 | |
*****************************************************************/
d ConvertToHex(Int NumToConvert)

 UInt    tmpNumber = 0;
 UInt    counter = 1;
 UInt    i = 1;

 StrCopy(gMessageToDisplay, "");                        //Clear String
 //StrCat(gMessageToDisplay, "<");                          //Start "<"
 do
    {
    if (counter == 1)(tmpNumber = NumToConvert/16);     //Calculate upper nibble
    else (tmpNumber = NumToConvert - (tmpNumber * 16)); //Calculate lower nibble
    switch (tmpNumber)                                  //Add appropriate values
        {
        case 0:
            StrCat(gMessageToDisplay, "0");
            break;
        case 1:
            StrCat(gMessageToDisplay, "1");
            break;
```

```
        case 2:
            StrCat(gMessageToDisplay, "2");
            break;
        case 3:
            StrCat(gMessageToDisplay, "3");
            break;
        case 4:
            StrCat(gMessageToDisplay, "4");
            break;
        case 5:
            StrCat(gMessageToDisplay, "5");
            break;
        case 6:
            StrCat(gMessageToDisplay, "6");
            break;
        case 7:
            StrCat(gMessageToDisplay, "7");
            break;
        case 8:
            StrCat(gMessageToDisplay, "8");
            break;
        case 9:
            StrCat(gMessageToDisplay, "9");
            break;
        case 10:
            StrCat(gMessageToDisplay, "A");
            break;
        case 11:
            StrCat(gMessageToDisplay, "B");
            break;
        case 12:
            StrCat(gMessageToDisplay, "C");
            break;
        case 13:
            StrCat(gMessageToDisplay, "D");
            break;
        case 14:
            StrCat(gMessageToDisplay, "E");
            break;
        case 15:
            StrCat(gMessageToDisplay, "F");
            break;
        #if(debug)
        default:
            StrCat(gMessageToDisplay, "0");
            break;
        #endif
        }//End Switch
    counter--;
    i++;
    } //End While
  while (counter  != -1);
    //StrCat(gMessageToDisplay, ">");                          //Close String
  //End Function ConvertToHex

agma mark ----------------


*****************************************************************************\
                                                                             ||
 FUNCTION:     MainFormInit                                                  ||
                                                                             ||
 DESCRIPTION: This routine initializes the MainForm form.                    ||
                                                                             ||
 PARAMETERS:   frm - pointer to the MainForm form.                           ||
                                                                             ||
 RETURNED:     nothing                                                       ||
                                                                             ||
 REVISION HISTORY:                                                           ||
                                                                             ||
*****************************************************************************/
d MainFormInit(FormPtr frmP)

  //Word          fldIndex;
  Word          ObjectIndex;

  //fldIndex = FrmGetObjectIndex(frmP, Main_TransmittedFromPalmField);  //Get field index
  //FrmSetFocus(frmP, fldIndex);                                        //Set focus to field

  //The following two lines hide the port open icon.
  //If you are connected to the serial port,
  //the icon is displayed, otherwise there is nothing there.
  ObjectIndex = FrmGetObjectIndex(frmP, Main_PortOpenBitMap);
  FrmHideObject(frmP, ObjectIndex);
```

```
*********************************************************************\
                                                                     ||
 FUNCTION:    MainFormDoCommand                                      ||
                                                                     ||
 DESCRIPTION: This routine performs the menu command specified.      ||
                                                                     ||
 PARAMETERS:  command  - menu item id                                ||
                                                                     ||
 RETURNED:    nothing                                                ||
                                                                     ||
 REVISION HISTORY:                                                   ||
                                                                     ||
*********************************************************************/
olean MainFormDoCommand(Word command)


  FormType     *FrmP;
  Boolean      handled = false;

  switch (command)
      {
      case Info_About:
          MenuEraseStatus (0);
          FrmP = FrmInitForm(About_Form);                    //Load About form
          FrmPopupForm(Help_Form);                           //Saves initiating form informati
 behind the loaded form
          FrmDoDialog(FrmP);                                 //Display and wair for "Ok"
          FrmDeleteForm(FrmP);                               //Delete form and return to Main_
rm
          handled = true;                                    //Tell Event Handler, the event w
 processed
          break;
      case Info_Help:
          MenuEraseStatus (0);
          FrmP = FrmInitForm(Help_Form);                     //Load Help form
          FrmPopupForm(Help_Form);                           //Saves initiating form informati
 behind the loaded form
          FrmDoDialog(FrmP);                                 //Display and wait for "Ok"
          FrmDeleteForm(FrmP);                               //Delete form and return to Main_
rm
          handled = true;                                    //Tell Event Handler, the event w
 processed
          break;
      case Options_OpenSerialPort:
          MenuEraseStatus (0);
          if (!(gSerialPortOpen)) OpenSerialPort();          //If not already open, open the s
ial port
          handled = true;                                    //Tell Event Handler, the event w
 processed
          break;
      case Options_CloseSerialPort:
          MenuEraseStatus (0);
          if (gSerialPortOpen) CloseSerialPort();            //If open, close the serial port
          handled = true;                                    //Tell Event Handler, the event w
 processed
          break;
      case Options_DefineTransmitString:
          MenuEraseStatus (0);
          //if (gSerialPortOpen) CloseSerialPort();              //If serial port open, close

          //FrmGotoForm(Text_Form);                              //Switch to text form
          handled = true;                                    //Tell Event Handler, the event w
 processed
          break;
      }
  return handled;


*********************************************************************\
                                                                     ||
 FUNCTION:    MainFormHandleEvent                                    ||
                                                                     ||
 DESCRIPTION: This routine is the event handler for the              ||
              "Main_Form" of this application.                       ||
                                                                     ||
 PARAMETERS:  eventP  - a pointer to an EventType structure          ||
                                                                     ||
 RETURNED:    true if the event has handle and should not be passed  ||
              to a higher level handler.                             ||
```

```
REVISION HISTORY:

*****************************************************************/
)lean MainFormHandleEvent(EventPtr eventP)

  Boolean     handled = false;
  FormPtr     frmP;
  FieldType   *FieldP;
  char        chrBuffer;

  frmP = FrmGetActiveForm();                                    //Get Pointer to Active Form
  switch (eventP->eType)
      {
      case menuEvent:                                           //Process menu item
          return MainFormDoCommand(eventP->data.menu.itemID);
          break;
      case frmOpenEvent:                                        //On Form Open
          FrmDrawForm ( frmP);                                  //Refresh Form
          MainFormInit( frmP);                                  //Initialize Form
          if (!gSerialPortOpen)
              {
                  Baud = 9600 ;
                  OpenSerialPort();               //Open serial port
              }
          handled = true;                                       //Tell Event Handler, the event w
 processed
          break;
      case ctlSelectEvent:                                      //Will Handle Item Select Events
              switch (eventP->data.ctlSelect.controlID)
              {
                      case Main_Text1PushButton:                          //Open port, send Transmi
3tring 1
                          if (gSerialPortOpen) CloseSerialPort();
                          Baud = 2400;
                          OpenSerialPort();
                          break;
                      case Main_Text2PushButton:                          //Open port, send Transmi
3tring 2
                          if (gSerialPortOpen) CloseSerialPort();
                          Baud = 9600;
                          OpenSerialPort();
                          break;
                      case Main_Text3PushButton:                          //Open port, send Transmi
3tring 3
                          if (gSerialPortOpen) CloseSerialPort();
                          Baud = 19200;
                          OpenSerialPort();
                          break;
                      case Main_Text4PushButton:                          //Open port, send Transmi
3tring 4
                          if (gSerialPortOpen) CloseSerialPort();
                          Baud = 38400;
                          OpenSerialPort();
                          break;
                      case Main_Text5PushButton:                          //Open port, send Transmi
3tring 5
                          if (gSerialPortOpen) CloseSerialPort();
                          Baud = 57600;
                          OpenSerialPort();
                          break;
                      case Main_HelpButton:                               //Displays Help Dialog
                          MenuEraseStatus (0);
                          frmP = FrmInitForm(Help_Form);                  //Load the Help form
                          FrmPopupForm(Help_Form);                        //Saves initiating form i
)rmation behind the loaded form
                          FrmDoDialog(frmP);                              //Display and wait for "O

                          FrmDeleteForm(frmP);                            //Delete the form (return
 Main_Form)
                          handled = true;
                          break;
                      case Main_ClearButton:                              //Clears the screen, both
)ceive and transmit
                          ClearScreenData(Main_ReceivedFromExternalField);
                          UpdateScrollBar();
                          break;
                      default:
                          break;
              }

              break;

          case keyDownEvent:
```

```
            if (eventP->data.keyDown.chr == pageUpChr ) //if it is a scroll up
            {
                Scroll (0)  ;//up
                handled = true ;
            }
            else if (eventP->data.keyDown.chr == pageDownChr) // if it is a scroll down
            {
                Scroll (1 )  ;//down
                handled = true ;
            }
            else  //if it is a character that is entered
            {
                frmP = FrmGetActiveForm ();
                FrmHandleEvent (frmP, eventP);
                UpdateScrollBar () ;
                handled = true ;
            }
        break;

        case sclRepeatEvent:  // P13. the scroll bar was pressed
            EditViewScroll (eventP->data.sclRepeat.newValue -
            eventP->data.sclRepeat.value);

            // Repeating controls don't repeat if handled is set true.
        break;

        default:
            break;
    }
 return handled;                                          //Tell Event Handler, whether eve
was handled


agma mark ----------------

*****************************************************************\
                                                                 ||
 FUNCTION:    TextFormInit                                       ||
                                                                 ||
 DESCRIPTION: Initializes the text form.                         ||
                                                                 ||
 PARAMETERS:  form - pointer to the text form.                   ||
                                                                 ||
 RETURNED:    nothing                                            ||
                                                                 ||
 REVISION HISTORY:                                               ||
                                                                 ||
*****************************************************************/
id TextFormInit(FormPtr FrmP)

  //Word  fldIndex;

  //fldIndex = FrmGetObjectIndex(FrmP, Text_Text1Field);      //Get field index
  //FrmSetFocus(FrmP, fldIndex);                             //Set focus to field



agma mark ----------------

*****************************************************************\
                                                                 ||
 FUNCTION:    GetObjectPtr                                       ||
                                                                 ||
 DESCRIPTION: This routine returns a pointer to an object in the ||
              current form.                                      ||
                                                                 ||
 PARAMETERS:  objectId - id of the object to retrieve            ||
                                                                 ||
 RETURNED:    VoidPtr - pointer to the object passed             ||
                                                                 ||
 REVISION HISTORY:                                               ||
                                                                 ||
*****************************************************************/
dPtr GetObjectPtr(Word objectID)
  {
  FormPtr frmP;


  frmP = FrmGetActiveForm();
  return (FrmGetObjectPtr(frmP, FrmGetObjectIndex(frmP, objectID)));
```

```
*********************************************************************\
                                                                     ||
FUNCTION:    RomVersionCompatible                                    ||
                                                                     ||
DESCRIPTION: Verifies that ROM version meets minimum requirement.    ||
                                                                     ||
PARAMETERS:  requiredVersion - minimum rom version required          ||
                               (see sysFtrNumROMVersion in           ||
                                SystemMgr.h for format)              ||
             launchFlags     - flags that indicate if the            ||
                               application UI is initialized.        ||
                                                                     ||
RETURNED:    error code or zero if rom is compatible                 ||
                                                                     ||
REVISION HISTORY:                                                    ||
                                                                     ||
*********************************************************************/
; RomVersionCompatible(DWord requiredVersion, Word launchFlags)

  DWord romVersion;

  // See if we're on in minimum required version of the ROM or later.
  FtrGet(sysFtrCreator, sysFtrNumROMVersion, &romVersion);
  if (romVersion < requiredVersion)
      {
      if ((launchFlags & (sysAppLaunchFlagNewGlobals | sysAppLaunchFlagUIApp)) ==
          (sysAppLaunchFlagNewGlobals | sysAppLaunchFlagUIApp))
          {
          FrmAlert (RomIncompatibleAlert);

          // Pilot 1.0 will continuously relaunch this app unless we switch to
          // another safe one.
          if (romVersion < sysMakeROMVersion(2,0,0,sysROMStageRelease,0))
              AppLaunchWithCommand(sysFileCDefaultApp, sysAppLaunchCmdNormalLaunch, NULL);
          }

      return (sysErrRomIncompatible);
      }

  return (0);



*********************************************************************\
                                                                     ||
FUNCTION:    AppHandleEvent                                          ||
                                                                     ||
DESCRIPTION: This routine loads form resources and set the event     ||
             handler for the form loaded.                            ||
                                                                     ||
PARAMETERS:  event  - a pointer to an EventType structure            ||
                                                                     ||
RETURNED:    true if the event has handle and should not be passed   ||
             to a higher level handler.                              ||
                                                                     ||
REVISION HISTORY:                                                    ||
                                                                     ||
*********************************************************************/
)lean AppHandleEvent( EventPtr eventP)

  Word formId;
  FormPtr frmP;


  if (eventP->eType == frmLoadEvent)
      {
      // Load the form resource.
      formId = eventP->data.frmLoad.formID;
      frmP = FrmInitForm(formId);
      FrmSetActiveForm(frmP);

      // Set the event handler for the form.  The handler of the currently
      // active form is called by FrmHandleEvent each time is receives an
      // event.
      switch (formId)
          {
          case Main_Form:
              FrmSetEventHandler(frmP, MainFormHandleEvent);
              break;
          default:
              break;
```

```
        }
      return true;
      }

  return false;


*****************************************************************\
                                                                 ||
 FUNCTION:    AppEventLoop                                       ||
                                                                 ||
 DESCRIPTION: This routine is the event loop for the application.||
                                                                 ||
 PARAMETERS:  nothing                                            ||
                                                                 ||
 RETURNED:    nothing                                            ||
                                                                 ||
 REVISION HISTORY:                                               ||
                                                                 ||
*****************************************************************/
id AppEventLoop(void)

  Word       error;
  EventType  event;
  DWord      ResetTimeOut;

  //Initialize Variables
  ResetTimeOut = TimGetSeconds();

  do {
            //To handle input received, since input received does not generate an event
            EvtGetEvent(&event, 50);                          //Generates an event

            //Prevent sleep mode, to handle incoming data,
            //call EvtResetAutoOffTimer every 45 seconds
            if (TimGetSeconds() - ResetTimeOut > 45)          //If time exceeds 45 seconds
arliest sleep at one minute)
              {
              EvtResetAutoOffTimer();                         //Reset timer
              ResetTimeOut = TimGetSeconds();
              }

            //Retrieve Data from Serial Port
            ReceiveData();

            //Handle Events
            if (! SysHandleEvent(&event))
              if (! MenuHandleEvent(0, &event, &error))
                if (! AppHandleEvent(&event))
                  FrmDispatchEvent(&event);


          if (event.eType == appStopEvent)
          {
              CloseSerialPort();
          }

    } while (event.eType != appStopEvent);


*****************************************************************\
                                                                 ||
 FUNCTION:     AppStart                                          ||
                                                                 ||
 DESCRIPTION:  Get the current application's preferences.        ||
                                                                 ||
 PARAMETERS:   nothing                                           ||
                                                                 ||
 RETURNED:     Err value 0 if nothing went wrong                 ||
                                                                 ||
 REVISION HISTORY:                                               ||
                                                                 ||
*****************************************************************/
r AppStart(void)


  gPrefsSize = sizeof(SerialCommPreferenceType);              //Set size of preferences

  gPrefs.Text1[0] = chrNull;                                  //Initialize preferences
  gPrefs.Text2[0] = chrNull;
```

```
  gPrefs.Text3[0] = chrNull;
  gPrefs.Text4[0] = chrNull;
  gPrefs.Text5[0] = chrNull;


  if (PrefGetAppPreferences(appFileCreator, appPrefID,
      &gPrefs, &gPrefsSize, true) == noPreferenceFound)       // Read the saved preferences / s
ed-state information.
      {
      gPrefs.Text1[0] = chrNull;                              //If No Preferences Found, fill w
h NULL
      gPrefs.Text2[0] = chrNull;
      gPrefs.Text3[0] = chrNull;
      gPrefs.Text4[0] = chrNull;
      gPrefs.Text5[0] = chrNull;
      }
  return 0;



*********************************************************************\
                                                                     ||
 FUNCTION:    AppStop                                               ||
                                                                     ||
 DESCRIPTION: Save the current state of the application.            ||
                                                                     ||
 PARAMETERS:  nothing                                               ||
                                                                     ||
 RETURNED:    nothing                                               ||
                                                                     ||
 REVISION HISTORY:                                                  ||
                                                                     ||
*********************************************************************/
id AppStop(void)

  if (gSerialPortOpen) CloseSerialPort();
  FrmCloseAllForms();

  // Write the saved preferences / saved-state information.  This data
  // will be backed up during a HotSync.


  gPrefsSize = sizeof(SerialCommPreferenceType);

  PrefSetAppPreferences (appFileCreator, appPrefID, appPrefVersionNum,
                         &gPrefs,  gPrefsSize, true);



*********************************************************************\
                                                                     ||
 FUNCTION:    SerialCommPilotMain                                   ||
                                                                     ||
 DESCRIPTION: This is the main entry point for the application.     ||
 PARAMETERS:  cmd - Word value specifying the launch code.          ||
              cmdPB - pointer to a structure that is associated with ||
                      the launch code.
              launchFlags -  Word value providing extra information ||
                             about the launch.                       ||
                                                                     ||
 RETURNED:    Result of launch                                      ||
                                                                     ||
 REVISION HISTORY:                                                  ||
                                                                     ||
*********************************************************************/
ord SerialCommPilotMain(Word cmd, Ptr cmdPBP, Word launchFlags)

  Err error;


  error = RomVersionCompatible (ourMinVersion, launchFlags);
  if (error) return (error);

  switch (cmd)
      {
      case sysAppLaunchCmdNormalLaunch:
          gSerialPortOpen = 0;
          error = AppStart();
          if (error)
              return error;
```

```
            FrmGotoForm(Main_Form);
            AppEventLoop();
            AppStop();
            break;

        default:
            break;

        }

    return 0;


*****************************************************************\
                                                                 ||
 FUNCTION:    PilotMain                                         ||
                                                                 ||
 DESCRIPTION: This is the main entry point for the application.  ||
                                                                 ||
 PARAMETERS:  cmd - Word value specifying the launch code.       ||
              cmdPB - pointer to a structure that is associated with ||
                      the launch code.                           ||
              launchFlags -  Word value providing extra information ||
                             about the launch.                   ||
                                                                 ||
 RETURNED:    Result of launch                                  ||
                                                                 ||
 REVISION HISTORY:                                               ||
                                                                 ||
                                                                 ||
*****************************************************************/
ord PilotMain( Word cmd, Ptr cmdPBP, Word launchFlags)


    return SerialCommPilotMain(cmd, cmdPBP, launchFlags);




*************************************************************************

atic void Scroll(int flag )
    {
    FieldPtr fld ;
    Short value;
    Short min;
    Short max;
    Short pageSize;
    Word linesToScroll;
    ScrollBarPtr bar;

    fld = GetObjectPtr (Main_ReceivedFromExternalField)  ;

    if ( FldScrollable ( fld , (DirectionType) flag ) ) // check if it is scrollable in the spec.
fied direction

    {
        linesToScroll = FldGetVisibleLines (fld) - 1;
        FldScrollField (fld, linesToScroll, (DirectionType)flag );

    // Update the scroll bar.
        bar = GetObjectPtr (Main_ScrollBarScrollBar);
        SclGetScrollBar (bar, &value, &min, &max, &pageSize);

        if (flag == 0)
            value -= linesToScroll;
        else
            value += linesToScroll;

        SclSetScrollBar (bar, value, min, max, pageSize);

    }


    return  ;
    }


/*************************************************************************

atic void UpdateScrollBar ()
```

```
  Word scrollPos;
  Word textHeight;
  Word fieldHeight;
  Short maxValue;
  FieldPtr fld;
  ScrollBarPtr bar;
  Word Len ;
  fld = GetObjectPtr ( Main_ReceivedFromExternalField );
  bar = GetObjectPtr (Main_ScrollBarScrollBar);

  Len = FldGetTextLength(fld);
  FldSetScrollPosition ( fld , Len);

  // get the values necessary to update the scroll bar.
  FldGetScrollValues (fld, &scrollPos, &textHeight,  &fieldHeight);

  if (textHeight > fieldHeight)
      maxValue = textHeight - fieldHeight;
  else if (scrollPos)
      maxValue = scrollPos;
  else
      maxValue = 0;


  SclSetScrollBar (bar, scrollPos, 0, maxValue, fieldHeight-1);




*****************************************************************************


atic void EditViewScroll (Short linesToScroll)

  Word                blankLines;
  Short               min;
  Short               max;
  Short               value;
  Short               pageSize;
  FieldPtr            fld;
  ScrollBarPtr        bar;

  fld = GetObjectPtr (Main_ReceivedFromExternalField);

  if (linesToScroll < 0)
      {
      blankLines = FldGetNumberOfBlankLines (fld);
      FldScrollField (fld, -linesToScroll, (DirectionType)0);

      // If there were blank lines visible at the end of the field
      // then we need to update the scroll bar.
      if (blankLines)
          {
          // Update the scroll bar.
          bar = GetObjectPtr (Main_ScrollBarScrollBar);
          SclGetScrollBar (bar, &value, &min, &max, &pageSize);
          if (blankLines > -linesToScroll)
              max += linesToScroll;
          else
              max -= blankLines;
          SclSetScrollBar (bar, value, min, max, pageSize);
          }
      }

  else if (linesToScroll > 0)
      FldScrollField (fld, linesToScroll, (DirectionType)1);



*****************************************************************************
```

```
  Header generated by Constructor for PalmOS 1.2

  Generated at 1:23:45 PM on Saturday, March 27, 2004

  Generated for file: D:\sem10\FYP\PALM Programs\Snooper\Src\Snooper.rsrc

  THIS IS AN AUTOMATICALLY GENERATED HEADER FILE FROM CONSTRUCTOR FOR PALMOS;
  - DO NOT EDIT - CHANGES MADE TO THIS FILE WILL BE LOST

  Palm App Name:       "Smart Card Snooper"

  Palm App Version:       "1.0"


  Resource: tFRM 1000
efine Main_Form                                 1000  //(Left Origin = 0, Top Origin = 0, Width
 158, Height = 160, Usable = 1, Modal = 0, Save Behind = 0, Help ID = 0, Menu Bar ID = 1000, De
ult Button ID = 0)
efine Main_ClearButton                          1008  //(Left Origin = 1, Top Origin = 149, Wid
 = 34, Height = 10, Usable = 1, Anchor Left = 1, Frame = 1, Non-bold Frame = 1, Font = Standard

efine Main_HelpButton                           1016  //(Left Origin = 142, Top Origin = 142, W
th = 16, Height = 16, Usable = 1, Anchor Left = 1, Frame = 1, Non-bold Frame = 1, Font = Standa
)
efine Main_HelpBitMap                           1010  //(Left Origin = 142, Top Origin = 142, B
map Resource ID = 1010, Usable = 1)
efine Main_PortOpenBitMap                       1011  //(Left Origin = 38, Top Origin = 144, Bi
ap Resource ID = 1011, Usable = 1)
efine Main_ReceivedFromExternalField            1002  //(Left Origin = 9, Top Origin = 29, Widt
= 140, Height = 104, Usable = 1, Editable = 1, Underline = 0, Single Line = 0, Dynamic Size = 1
Left Justified = 1, Max Characters = 60000, Font = Standard, Auto Shift = 0, Has Scroll Bar = 0
Numeric = 0)
efine Main_MainIndicatorGrafittiShift           1007  //(Left Origin = 131, Top Origin = 148)
efine Main_ReceivedLabel                        1001  //(Left Origin = 0, Top Origin = 17, Usab
 = 1, Font = Standard)
efine Main_Text1PushButton                      1020  //(Left Origin = 71, Top Origin = 132, Wi
h = 22, Height = 12, Usable = 1, Group ID = 1, Font = Standard)
efine Main_Text2PushButton                      1021  //(Left Origin = 96, Top Origin = 132, Wi
h = 22, Height = 12, Usable = 1, Group ID = 1, Font = Standard)
efine Main_Text3PushButton                      1022  //(Left Origin = 58, Top Origin = 147, Wi
h = 24, Height = 12, Usable = 1, Group ID = 1, Font = Standard)
efine Main_Text4PushButton                      1023  //(Left Origin = 85, Top Origin = 147, Wi
h = 24, Height = 12, Usable = 1, Group ID = 1, Font = Standard)
efine Main_Text5PushButton                      1024  //(Left Origin = 112, Top Origin = 147, W
th = 24, Height = 12, Usable = 1, Group ID = 1, Font = Standard)
efine Main_ScrollBarScrollBar                   1005  //(Left Origin = 151, Top Origin = 24, Wi
h = 7, Height = 109, Usable = 1, Value = 0, Minimum Value = 0, Maximum Value = 0, Page Size = 0

efine SmartCardSnooperFormGroupID               1


  Resource: tFRM 1100
efine About_Form                                1100  //(Left Origin = 0, Top Origin = 0, Width
 160, Height = 160, Usable = 1, Modal = 1, Save Behind = 1, Help ID = 0, Menu Bar ID = 0, Defau
 Button ID = 0)
efine About_Unnamed1113Button                   1113  //(Left Origin = 120, Top Origin = 143, W
th = 36, Height = 12, Usable = 1, Anchor Left = 1, Frame = 1, Non-bold Frame = 1, Font = Standa

efine About_CompLLBitMap                        1006  //(Left Origin = 73, Top Origin = 79, Bit
p Resource ID = 1006, Usable = 1)
efine About_PalmBitMap                          1000  //(Left Origin = 56, Top Origin = 96, Bit
p Resource ID = 1000, Usable = 1)
efine About_CableBitMap                         1001  //(Left Origin = 56, Top Origin = 80, Bit
p Resource ID = 1001, Usable = 1)
efine About_CompLRBitMap                        1007  //(Left Origin = 89, Top Origin = 79, Bit
p Resource ID = 1007, Usable = 1)
efine About_CompULBitMap                        1002  //(Left Origin = 73, Top Origin = 48, Bit
p Resource ID = 1002, Usable = 1)
efine About_CompURBitMap                        1003  //(Left Origin = 89, Top Origin = 48, Bit
p Resource ID = 1003, Usable = 1)
efine About_CompMLBitMap                        1004  //(Left Origin = 73, Top Origin = 63, Bit
p Resource ID = 1004, Usable = 1)
efine About_CompMRBitMap                        1005  //(Left Origin = 89, Top Origin = 63, Bit
p Resource ID = 1005, Usable = 1)
efine About_AppTitleLabel                       1101  //(Left Origin = 2, Top Origin = 15, Usab
 = 1, Font = Bold)
efine About_VersionLabel                        1102  //(Left Origin = 2, Top Origin = 32, Usab
 = 1, Font = Standard)
efine About_CopyrightLabel                      1103  //(Left Origin = 2, Top Origin = 120, Usa
e = 1, Font = Standard)
efine About_CompanyInfoLabel                    1104  //(Left Origin = 1, Top Origin = 135, Usa
e = 1, Font = Standard)
```

```
  Resource: tFRM 1200
efine Help_Form                                 1200  //(Left Origin = 0, Top Origin = 0, Width
 160, Height = 160, Usable = 1, Modal = 1, Save Behind = 1, Help ID = 0, Menu Bar ID = 0, Defau
 Button ID = 0)
efine Help_Unnamed1203Button                    1203  //(Left Origin = 120, Top Origin = 144, W
:h = 36, Height = 12, Usable = 1, Anchor Left = 1, Frame = 1, Non-bold Frame = 1, Font = Standa
 )
efine Help_HelpLabel                            1204  //(Left Origin = 0, Top Origin = 13, Usab
 = 1, Font = Standard)


  Resource: Talt 1001
efine RomIncompatibleAlert                      1001
efine RomIncompatibleOK                         0

  Resource: Talt 1000
efine SerialManagerAlert                        1000
efine SerialManagerOK                           0

  Resource: Talt 1002
efine OpenPortAlert                             1002
efine OpenPortOK                                0

  Resource: Talt 1003
efine CheckPortAlert                            1003
efine CheckPortOK                               0

  Resource: Talt 1004
efine PortBusyAlert                             1004
efine PortBusyOK                                0

  Resource: Talt 1005
efine DataTransmitAlert                         1005
efine DataTransmitOK                            0

  Resource: Talt 1006
efine PortTimeoutAlert                          1006
efine PortTimeoutOK                             0

  Resource: Talt 1007
efine ClosePortAlert                            1007
efine ClosePortOK                               0

  Resource: Talt 1008
efine NoDataToSendAlert                         1008
efine NoDataToSendOK                            0

  Resource: Talt 1100
efine DebugAlert                                1100
efine DebugOK                                   0

  Resource: Talt 1009
efine CommSettingsAlert                         1009
efine CommSettingsOK                            0

  Resource: Talt 1010
efine MemoryAlert                               1010
efine MemoryOK                                  0

  Resource: MBAR 1000
efine Main_MenuBar                              1000

  Resource: MENU 1100
efine Options_Menu                              1100
efine Options_OpenSerialPort                    1100  //  Command Key: O
efine Options_CloseSerialPort                   1101  //  Command Key: C
efine Options_DefineTransmitString              1103  //  Command Key: D

  Resource: MENU 1200
efine Info_Menu                                 1200
efine Info_About                                1200  //  Command Key: A
efine Info_Help                                 1201  //  Command Key: H

  Resource: tSTR 1000
efine HelpString                                1000  // "Close your serial port when not in us
 the serial port will drain your battery.  Use Define Transmit String to send hex numbers.  Enc
se hex numbers in <  > brackets, and send one byte at a time, eg.: carriage return & linefeed =
13><10> "

  Resource: PICT 1005
efine CompMRBitmap                              1005
  Resource: PICT 1004
efine CompMLBitmap                              1004
  Resource: PICT 1003
efine CompURBitmap                              1003
```

```
  Resource: PICT 1002
efine CompULBitmap                      1002
  Resource: PICT 1001
efine CableBitmap                       1001
  Resource: PICT 1000
efine PalmBitmap                        1000
  Resource: PICT 1006
efine CompLLBitmap                      1006
  Resource: PICT 1007
efine CompLRBitmap                      1007
  Resource: PICT 1010
efine HelpBitmap                        1010
  Resource: PICT 1011
efine PortOpenBitmap                    1011
```

Appendix O

Complete Source Code for the Visual C++ Snooping Program

```
 SerialApp.cpp : Defines the class behaviors for the application.


nclude "stdafx.h"
nclude "SerialApp.h"
nclude "serialCtl.hpp"
nclude "SerialAppDlg.h"

fdef _DEBUG
efine new DEBUG_NEW
ndef THIS_FILE
atic char THIS_FILE[] = __FILE__;
ndif

/////////////////////////////////////////////////////////////////////////////
 CSerialAppApp

GIN_MESSAGE_MAP(CSerialAppApp, CWinApp)
  //{{AFX_MSG_MAP(CSerialAppApp)
      // NOTE - the ClassWizard will add and remove mapping macros here.
      //    DO NOT EDIT what you see in these blocks of generated code!
  //}}AFX_MSG
  ON_COMMAND(ID_HELP, CWinApp::OnHelp)
D_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
 CSerialAppApp construction

erialAppApp::CSerialAppApp()

  // TODO: add construction code here,
  // Place all significant initialization in InitInstance


/////////////////////////////////////////////////////////////////////////////
 The one and only CSerialAppApp object

erialAppApp theApp;

/////////////////////////////////////////////////////////////////////////////
 CSerialAppApp initialization

OL CSerialAppApp::InitInstance()

  AfxEnableControlContainer();

  // Standard initialization
  // If you are not using these features and wish to reduce the size
  //  of your final executable, you should remove from the following
  //  the specific initialization routines you do not need.

:def _AFXDLL
  Enable3dControls();          // Call this when using MFC in a shared DLL
.se
  Enable3dControlsStatic();    // Call this when linking to MFC statically
ıdif

  CSerialAppDlg dlg;
  m_pMainWnd = &dlg;
  int nResponse = dlg.DoModal();
  if (nResponse == IDOK)
  {
      // TODO: Place code here to handle when the dialog is
      //  dismissed with OK
  }
  else if (nResponse == IDCANCEL)
  {
      // TODO: Place code here to handle when the dialog is
      //  dismissed with Cancel
  }

  // Since the dialog has been closed, return FALSE so that we exit the
  //  application, rather than start the application's message pump.
  return FALSE;
```

```
 SerialAppDlg.cpp : implementation file


nclude "stdafx.h"
nclude "serialCtl.hpp"
nclude "serialThread.hpp"
nclude "SerialApp.h"
nclude "SerialAppDlg.h"

fdef _DEBUG
efine new DEBUG_NEW
ndef THIS_FILE
atic char THIS_FILE[] = __FILE__;
ndif

/////////////////////////////////////////////////////////////////////////////
 CAboutDlg dialog used for App About

ass CAboutDlg : public CDialog

blic:
  CAboutDlg();

 Dialog Data
  //{{AFX_DATA(CAboutDlg)
  enum { IDD = IDD_ABOUTBOX };
  //}}AFX_DATA

  // ClassWizard generated virtual function overrides
  //{{AFX_VIRTUAL(CAboutDlg)
  protected:
  virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
  //}}AFX_VIRTUAL

 Implementation
otected:
  //{{AFX_MSG(CAboutDlg)
  //}}AFX_MSG
  DECLARE_MESSAGE_MAP()


outDlg::CAboutDlg() : CDialog(CAboutDlg::IDD)

  //{{AFX_DATA_INIT(CAboutDlg)
  //}}AFX_DATA_INIT


id CAboutDlg::DoDataExchange(CDataExchange* pDX)

  CDialog::DoDataExchange(pDX);
  //{{AFX_DATA_MAP(CAboutDlg)
  //}}AFX_DATA_MAP


GIN_MESSAGE_MAP(CAboutDlg, CDialog)
  //{{AFX_MSG_MAP(CAboutDlg)
      // No message handlers
  //}}AFX_MSG_MAP
)_MESSAGE_MAP()

'/////////////////////////////////////////////////////////////////////////////
 CSerialAppDlg dialog

erialAppDlg::CSerialAppDlg(CWnd* pParent /*=NULL*/)
  : CDialog(CSerialAppDlg::IDD, pParent)

  //{{AFX_DATA_INIT(CSerialAppDlg)
  m_namePort = _T("");
  m_baudRate = _T("");
  m_monitorRec = _T("");
  m_monitorSend = _T("");
  m_status_port = _T("");
penPortActivate = false;
:losePortActivate = false;
endActivate = false;
ctiveProccess = FALSE;
  //}}AFX_DATA_INIT
  // Note that LoadIcon does not require a subsequent DestroyIcon in Win32
  m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
  m_baudRate = "9600";
  m_namePort = "COM1";


d CSerialAppDlg::DoDataExchange(CDataExchange* pDX)
```

```
  CDialog::DoDataExchange(pDX);
  //{{AFX_DATA_MAP(CSerialAppDlg)
  DDX_Control(pDX, IDC_MONITOR_REC, m_DISPLAY);
  DDX_CBString(pDX, IDC_NAME_PORT, m_namePort);
  DDX_CBString(pDX, IDC_BOUDRATE, m_baudRate);
  DDX_Text(pDX, IDC_MONITOR_REC, m_monitorRec);
  DDX_Text(pDX, IDC_MONITOR_SEND, m_monitorSend);
  DDX_Text(pDX, IDC_STATUS_PORT, m_status_port);
  //}}AFX_DATA_MAP


GIN_MESSAGE_MAP(CSerialAppDlg, CDialog)
  //{{AFX_MSG_MAP(CSerialAppDlg)
  ON_WM_SYSCOMMAND()
  ON_WM_PAINT()
  ON_WM_QUERYDRAGICON()
  ON_BN_CLICKED(IDC_OPEN_PORT, OnOpenPort)
  ON_BN_CLICKED(IDC_CLOSE_PORT, OnClosePort)
  ON_BN_CLICKED(IDC_EXIT, OnExit)
  ON_BN_CLICKED(IDC_SEND_DATA, OnSendData)
  ON_BN_CLICKED(IDC_Clear, OnClear)
  //}}AFX_MSG_MAP
D_MESSAGE_MAP()

////////////////////////////////////////////////////////////////////////////
 CSerialAppDlg message handlers

)L CSerialAppDlg::OnInitDialog()

  CDialog::OnInitDialog();

  // Add "About..." menu item to system menu.

  // IDM_ABOUTBOX must be in the system command range.
  ASSERT((IDM_ABOUTBOX & 0xFFF0) == IDM_ABOUTBOX);
  ASSERT(IDM_ABOUTBOX < 0xF000);

  CMenu* pSysMenu = GetSystemMenu(FALSE);
  if (pSysMenu != NULL)
  {
      CString strAboutMenu;
      strAboutMenu.LoadString(IDS_ABOUTBOX);
      if (!strAboutMenu.IsEmpty())
      {
          pSysMenu->AppendMenu(MF_SEPARATOR);
          pSysMenu->AppendMenu(MF_STRING, IDM_ABOUTBOX, strAboutMenu);
      }
  }

  // Set the icon for this dialog.  The framework does this automatically
  //  when the application's main window is not a dialog
  SetIcon(m_hIcon, TRUE);          // Set big icon
  SetIcon(m_hIcon, FALSE);         // Set small icon

  // TODO: Add extra initialization here
 etDlgItem(IDC_CLOSE_PORT)->EnableWindow(FALSE);
  serialProcess =
  (SerialThread*)AfxBeginThread(RUNTIME_CLASS(SerialThread),
  THREAD_PRIORITY_NORMAL, 0, CREATE_SUSPENDED);
 erialProcess->setOwner(this);
  return TRUE;  // return TRUE  unless you set the focus to a control


 d CSerialAppDlg::OnSysCommand(UINT nID, LPARAM lParam)

  if ((nID & 0xFFF0) == IDM_ABOUTBOX)
  {
      CAboutDlg dlgAbout;
      dlgAbout.DoModal();
  }
  else
  {
      CDialog::OnSysCommand(nID, lParam);
  }


 If you add a minimize button to your dialog, you will need the code below
  to draw the icon.  For MFC applications using the document/view model,
  this is automatically done for you by the framework.

 i CSerialAppDlg::OnPaint()

  if (IsIconic())
  {
      CPaintDC dc(this); // device context for painting
```

```
        SendMessage(WM_ICONERASEBKGND, (WPARAM) dc.GetSafeHdc(), 0);

        // Center icon in client rectangle
        int cxIcon = GetSystemMetrics(SM_CXICON);
        int cyIcon = GetSystemMetrics(SM_CYICON);
        CRect rect;
        GetClientRect(&rect);
        int x = (rect.Width() - cxIcon + 1) / 2;
        int y = (rect.Height() - cyIcon + 1) / 2;

        // Draw the icon
        dc.DrawIcon(x, y, m_hIcon);
    }
    else
    {
        CDialog::OnPaint();
    }


 The system calls this to obtain the cursor to display while the user drags
    the minimized window.
URSOR CSerialAppDlg::OnQueryDragIcon()

    return (HCURSOR) m_hIcon;


id CSerialAppDlg::OnOpenPort()

    // TODO: Add your control notification handler code here
UpdateData(TRUE);
UpdateConfig();
openPortActivate = true;
closePortActivate = false;
activeProccess = TRUE;
UpdateData(TRUE);
serialProcess->ResumeThread();
GetDlgItem(IDC_CLOSE_PORT)->EnableWindow(TRUE);
GetDlgItem(IDC_OPEN_PORT)->EnableWindow(FALSE);
JpdateData(FALSE);


id CSerialAppDlg::OnClosePort()

    // TODO: Add your control notification handler code here
// Set signal of closing port serial communication.
closePortActivate =  true;
openPortActivate = false;
GetDlgItem(IDC_CLOSE_PORT)->EnableWindow(FALSE);
GetDlgItem(IDC_OPEN_PORT)->EnableWindow(TRUE);
JpdateData(FALSE);


id CSerialAppDlg::OnExit()

    // TODO: Add your control notification handler code here
// Set signal of closing port serial communication.
serialProcess->SuspendThread();
    this->DestroyWindow();


d CSerialAppDlg::OnSendData()

    // TODO: Add your control notification handler code here
// Set signal to send data of serial communication.
JpdateData(TRUE);
sendActivate = true;


d CSerialAppDlg::UpdateConfig()

// constant parameter.
configSerial_.ByteSize = 8;
configSerial_.StopBits = TWOSTOPBITS;
configSerial_.Parity = EVENPARITY;

switch(atoi(m_baudRate))

ase 110:
    configSerial_.BaudRate = CBR_110;
    break;
ase 300:
    configSerial_.BaudRate = CBR_300;
    break;
ase 600:
```

```
  configSerial_.BaudRate = CBR_600;
  break;
case 1200:
  configSerial_.BaudRate = CBR_1200;
  break;
case 2400:
  configSerial_.BaudRate = CBR_2400;
  break;
case 4800:
  configSerial_.BaudRate = CBR_4800;
  break;
case 9600:
  configSerial_.BaudRate = CBR_9600;
  break;
case 14400:
  configSerial_.BaudRate = CBR_14400;
  break;
case 19200:
  configSerial_.BaudRate = CBR_19200;
  break;
case 38400:
  configSerial_.BaudRate = CBR_38400;
  break;
case 56000:
  configSerial_.BaudRate = CBR_56000;
  break;
case 57600:
  configSerial_.BaudRate = CBR_57600;
  break;
case 115200 :
  configSerial_.BaudRate = CBR_115200;
  break;
case 128000:
  configSerial_.BaudRate = CBR_128000;
  break;
case 256000:
  configSerial_.BaudRate = CBR_256000;
  break;
default:
  break;
}


id CSerialAppDlg::OnClear()


  m_monitorRec = "" ;
  UpdateData(FALSE);
```

```
 stdafx.cpp : source file that includes just the standard includes
   SerialApp.pch will be the pre-compiled header
   stdafx.obj will contain the pre-compiled type information

nclude "stdafx.h"
```

```
{{NO_DEPENDENCIES}}
 Microsoft Developer Studio generated include file.
 Used by SerialApp.rc

efine IDM_ABOUTBOX                      0x0010
efine IDD_ABOUTBOX                      100
efine IDS_ABOUTBOX                      101
efine IDD_SERIALAPP_DIALOG              102
efine IDR_MAINFRAME                     128
efine IDR_ACCELERATOR1                  130
efine IDB_BITMAP1                       135
efine IDC_OPEN_PORT                     1000
efine IDC_CLOSE_PORT                    1001
efine IDC_EXIT                          1002
efine IDC_MONITOR_REC                   1003
efine IDC_MONITOR_SEND                  1004
efine IDC_SEND_DATA                     1005
efine IDC_NAME_PORT                     1007
efine IDC_BOUDRATE                      1008
efine IDC_STATUS_PORT                   1009
efine IDC_Clear                         1011

 Next default values for new objects

fdef APSTUDIO_INVOKED
fndef APSTUDIO_READONLY_SYMBOLS
efine _APS_NEXT_RESOURCE_VALUE          136
efine _APS_NEXT_COMMAND_VALUE           32777
efine _APS_NEXT_CONTROL_VALUE           1012
efine _APS_NEXT_SYMED_VALUE             101
ndif
ndif
```

```
 SerialApp.h : main header file for the SERIALAPP application


f !defined(AFX_SERIALAPP_H__361F5FC9_B80B_4224_805D_20EA4F9314AC__INCLUDED_)
efine AFX_SERIALAPP_H__361F5FC9_B80B_4224_805D_20EA4F9314AC__INCLUDED_

f _MSC_VER > 1000
ragma once
ndif // _MSC_VER > 1000

fndef __AFXWIN_H__
  #error include 'stdafx.h' before including this file for PCH
ndif

nclude "resource.h"       // main symbols

/////////////////////////////////////////////////////////////////////////////
 CSerialAppApp:
 See SerialApp.cpp for the implementation of this class


ass CSerialAppApp : public CWinApp

blic:
  CSerialAppApp();

 Overrides
  // ClassWizard generated virtual function overrides
  //{{AFX_VIRTUAL(CSerialAppApp)
  public:
  virtual BOOL InitInstance();
  //}}AFX_VIRTUAL

 Implementation

  //{{AFX_MSG(CSerialAppApp)
      // NOTE - the ClassWizard will add and remove member functions here.
      //    DO NOT EDIT what you see in these blocks of generated code !
  //}}AFX_MSG
  DECLARE_MESSAGE_MAP()



/////////////////////////////////////////////////////////////////////////////

{{AFX_INSERT_LOCATION}}
 Microsoft Visual C++ will insert additional declarations immediately before the previous line.

ndif // !defined(AFX_SERIALAPP_H__361F5FC9_B80B_4224_805D_20EA4F9314AC__INCLUDED_)
```

```
 SerialAppDlg.h : header file


f !defined(AFX_SERIALAPPDLG_H__81B8D820_84F4_495C_A799_9659B5277C70__INCLUDED_)
lefine AFX_SERIALAPPDLG_H__81B8D820_84F4_495C_A799_9659B5277C70__INCLUDED_

f _MSC_VER > 1000
ragma once
ndif // _MSC_VER > 1000

/////////////////////////////////////////////////////////////////////////////
 CSerialAppDlg dialog
ass SerialThread;
ass CSerialAppDlg : public CDialog

 Construction
blic:
  CSerialAppDlg(CWnd* pParent = NULL);    // standard constructor

 Dialog Data
  //{{AFX_DATA(CSerialAppDlg)
  enum { IDD = IDD_SERIALAPP_DIALOG };
  CEdit   m_DISPLAY;
  CString m_namePort;
  CString m_baudRate;
  CString m_monitorRec;
DCB configSerial_;
  CString m_monitorSend;
bool openPortActivate;
bool closePortActivate;
bool sendActivate;
  CString m_status_port;
SerialThread* serialProcess;
BOOL activeProccess;
  //}}AFX_DATA

  // ClassWizard generated virtual function overrides
  //{{AFX_VIRTUAL(CSerialAppDlg)
  protected:
  virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
  //}}AFX_VIRTUAL

 Implementation
otected:
  HICON m_hIcon;

  // Generated message map functions
  //{{AFX_MSG(CSerialAppDlg)
  virtual BOOL OnInitDialog();
  afx_msg void OnSysCommand(UINT nID, LPARAM lParam);
  afx_msg void OnPaint();
  afx_msg HCURSOR OnQueryDragIcon();
  afx_msg void OnOpenPort();
  afx_msg void OnClosePort();
  afx_msg void OnExit();
  afx_msg void OnSendData();
  afx_msg void OnClear();
  //}}AFX_MSG
  DECLARE_MESSAGE_MAP()
ivate:
  void UpdateConfig();


{{AFX_INSERT_LOCATION}}
 Microsoft Visual C++ will insert additional declarations immediately before the previous line.

ndif // !defined(AFX_SERIALAPPDLG_H__81B8D820_84F4_495C_A799_9659B5277C70__INCLUDED_)
```

```
 stdafx.h : include file for standard system include files,
  or project specific include files that are used frequently, but
      are changed infrequently


f !defined(AFX_STDAFX_H__C4A22ECB_4372_4960_A828_4B794461B5C3__INCLUDED_)
efine AFX_STDAFX_H__C4A22ECB_4372_4960_A828_4B794461B5C3__INCLUDED_

f _MSC_VER > 1000
ragma once
ndif // _MSC_VER > 1000

efine VC_EXTRALEAN        // Exclude rarely-used stuff from Windows headers

nclude <afxwin.h>         // MFC core and standard components
nclude <afxext.h>         // MFC extensions
nclude <afxdisp.h>        // MFC Automation classes
nclude <afxdtctl.h>       // MFC support for Internet Explorer 4 Common Controls
fndef _AFX_NO_AFXCMN_SUPPORT
nclude <afxcmn.h>         // MFC support for Windows Common Controls
ndif // _AFX_NO_AFXCMN_SUPPORT


{{AFX_INSERT_LOCATION}}
 Microsoft Visual C++ will insert additional declarations immediately before the previous line.

ndif // !defined(AFX_STDAFX_H__C4A22ECB_4372_4960_A828_4B794461B5C3__INCLUDED_)
```

```
*********************************************************************

filename:   D:\Applicaiton testing\SerialApp\serialCtl\SerialCtl.hpp
file path:  D:\Applicaiton testing\SerialApp\serialCtl
file base:  SerialCtl
file ext: hpp
author:   Dimitri Denamany.

purpose:  Use to control the serial communication's signal.
*********************************************************************/
nclude "afxwin.h"
nclude "serialCtl.hpp"
nclude "SerialApp.h"
nclude "SerialAppDlg.h"

 SerialCtl::SerialCtl()
--------------------------------------------------------------------------------
 Description: Constructor

rialCtl::SerialCtl()
tatusPort_(FALSE),
andlePort_(NULL)

// default parameter.
config_.ByteSize = 8;             // Byte of the Data.
config_.StopBits = ONESTOPBIT;    // Use one bit for stopbit.
config_.Parity = NOPARITY;        // No parity bit
config_.BaudRate = CBR_9600;      // Buadrate 9600 bit/sec


 SerialCtl::~SerialCtl()
--------------------------------------------------------------------------------
 Description: Destructor

rialCtl::~SerialCtl()

andlePort_ = NULL;


 SerialCtl::openPort(DCB dcb, const char* portName)
--------------------------------------------------------------------------------
 Description: Open the serial communication port by calling CreateFile
              function is as the API function. The dcb is a argument
              that contain the serial communication configuration.
              The portname is as name of device that want to open and perform.

)L
rialCtl::openPort(DCB dcb, const char* portName)

/ TODO: Add your control notification handler code here
f (statusPort_ == false)  // if port is opened already, not open port again.
{
  handlePort_ = CreateFile(portName,  // Specify port device: default "COM1"
  GENERIC_READ | GENERIC_WRITE,       // Specify mode that open device.
  0,                                  // the devide isn't shared.
  NULL,                               // the object gets a default security.
  OPEN_EXISTING,                      // Specify which action to take on file.
  0,                                  // default.
  NULL);                              // default.

  // Get current configuration of serial communication port.
  if (GetCommState(handlePort_,&config_) == 0)
  {
    AfxMessageBox("Get configuration port has problem.");
    return FALSE;
  }
  // Assign user parameter.
  config_.BaudRate = CBR_9600;    // Specify baud rate of communicaiton.
  config_.StopBits = TWOSTOPBITS;    // Specify stopbit of communication.
  config_.Parity = EVENPARITY;        // Specify parity of communication.
  config_.ByteSize = 8;    // Specify  byte of size of communication.


  // Set current configuration of serial communication port.
  if (SetCommState(handlePort_,&config_) == 0)
  {
     AfxMessageBox("Set configuration port has problem.");
     return FALSE;
  }

  // instance an object of COMMTIMEOUTS.
  COMMTIMEOUTS comTimeOut;
  // Specify time-out between charactor for receiving.
  comTimeOut.ReadIntervalTimeout = 5;
  // Specify value that is multiplied
```

```
  // by the requested number of bytes to be read.
  comTimeOut.ReadTotalTimeoutMultiplier = 5;
  // Specify value is added to the product of the
  // ReadTotalTimeoutMultiplier member
  comTimeOut.ReadTotalTimeoutConstant = 5;
  // Specify value that is multiplied
  // by the requested number of bytes to be sent.
  comTimeOut.WriteTotalTimeoutMultiplier = 5;
  // Specify value is added to the product of the
  // WriteTotalTimeoutMultiplier member
  comTimeOut.WriteTotalTimeoutConstant = 5;
  // set the time-out parameter into device control.
  SetCommTimeouts(handlePort_,&comTimeOut);
  // Updata port's status.


  statusPort_ = TRUE;
  return TRUE;
}
return FALSE;


 SerialCtl::closePort()
-------------------------------------------------------------------------
 Description: close communication by destroy handle of communication.

OL
rialCtl::closePort()

if (statusPort_ == TRUE)                // Port need to be open before.
{
  statusPort_ = false;                  // Update status
  if(CloseHandle(handlePort_) == 0)     // Call this function to close port.
  {
    AfxMessageBox("Port Closeing isn't successed.");
    return FALSE;
  }
  return TRUE;
}
return FALSE;


 read_scc(char* inputData,unsigned int sizeBuffer,unsigned int lengh)
-------------------------------------------------------------------------
 Description: read data from serial communication port.

OL
rialCtl::read_scc(char* inputData,
                  const unsigned int& sizeBuffer,
                  unsigned long& length)

if (ReadFile(handlePort_,  // handle of file to read
  inputData,               // handle of file to read
  sizeBuffer,              // number of bytes to read
  &length,                 // pointer to number of bytes read
  NULL) == 0)              // pointer to structure for data
{
  AfxMessageBox("Reading of serial communication has problem.");
  return FALSE;
}
if (length > 0)
{
  inputData[length] = NULL; // Assign end flag of message.
  return TRUE;
}
return TRUE;



 SerialCtl::write_scc(const char* outputData,
                  const unsigned int& sizeBuffer,
                  unsigned long& length)
-------------------------------------------------------------------------
 Description: write the data to serial communicaiton.

OL
rialCtl::write_scc(LPCVOID outputData,
                   const unsigned int& sizeBuffer,
                   unsigned long& length)

if (length > 0)
{
  if (WriteFile(handlePort_, // handle to file to write to
    outputData,              // pointer to data to write to file
    sizeBuffer,              // number of bytes to write
```

```
    &length,NULL) == 0)       // pointer to number of bytes written
  {
    AfxMessageBox("Reading of serial communication has problem.");
    return FALSE;
  }
  return TRUE;
}
return FALSE;


 SerialCtl::getStatusPort()
--------------------------------------------------------------------------------
 Description: the entry point to get port's status.

OL
rialCtl::getStatusPort()

return statusPort_;


**************************End of file*************************************/
```

```
*********************************************************************

 filename:   D:\Applicaiton testing\SerialApp\serialCtl\SerialCtl.hpp
 file path:  D:\Applicaiton testing\SerialApp\serialCtl
 file base:  SerialCtl
 file ext: hpp
 author:   Dimitri Denamany.

 purpose:  Use to control the serial communication's signal.
*********************************************************************/
.nclude "afxwin.h"
.nclude "serialCtl.hpp"
.nclude "resource.h"
.nclude "serialAppDlg.h"
.nclude "serialThread.hpp"

)nst unsigned short MAX_MESSAGE = 100;

IPLEMENT_DYNCREATE(SerialThread,CWinThread)
 SerialThread::SerialThread()
 ------------------------------------------------------------------------
 Constructor

 rialThread::SerialThread()
 trDlg(NULL)



 SerialThread::~SerialThread()
 ------------------------------------------------------------------------
 Deconstructor

 rialThread::~SerialThread()

 ptrDlg = NULL;


 SerialThread::InitInstance()
 ------------------------------------------------------------------------
 Deconstructor

 )L
 rialThread::InitInstance()

 return TRUE;


  SerialThread::Run()
 ------------------------------------------------------------------------
  Description: This is a virtual function that is called when thread process
               is created to be one task.

 :
 :ialThread::Run()

 '/ Check signal controlling and status to open serial communication.
 /hile(1)

   // Start process of serial communication operation.
   while(ptrDlg->activeProccess == TRUE)
   {
     // enter if there is command of openning and port has be closed before.
     if ((SCC::serialCtl().getStatusPort() == FALSE) &&
       ptrDlg->openPortActivate)
     {
       // open port by calling api function of class serialCtl.
       if (SCC::serialCtl().openPort(ptrDlg->configSerial_,
         ptrDlg->m_namePort) == TRUE)
       {
         // Indicate message to status moditor that commnication connected already.
         ptrDlg->SetDlgItemText(IDC_STATUS_PORT,"Connected");
       }
       else
       {
         // Have problem since opening serial communication.
         ptrDlg->activeProccess = FALSE;
       }
     }
     else if (ptrDlg->openPortActivate)
     {
       char mess[MAX_MESSAGE];
       unsigned int lenBuff = MAX_MESSAGE;
       unsigned long lenMessage;
       static CString outPut;
       if (SCC::serialCtl().read_scc(mess,lenBuff,lenMessage) == TRUE)
```

```
      {
        if (lenMessage > 0)
        {
          LPTSTR lpTempAPDU= new char [2*lenMessage+1];

          //this is the function to convert the bytes into its ASCII representation so that it
n be displayed
          ByteArrayToHexString((PBYTE)mess, lenMessage ,(char*)lpTempAPDU );

          ptrDlg->GetDlgItemText(IDC_MONITOR_REC,outPut);

          outPut = outPut + "\r\n" + lpTempAPDU;

          ptrDlg->SetDlgItemText(IDC_MONITOR_REC,outPut);

          delete [] lpTempAPDU ;

          // these are lines added so that the display dialog box would browse down by itself
          //without the user`````````` having to pull the scroll bar every time a byte is recei
d
          int NumLine =ptrDlg->m_DISPLAY.GetLineCount();
          ptrDlg->m_DISPLAY.GetLineCount();
          ptrDlg->m_DISPLAY.LineScroll(NumLine);
          ptrDlg->m_DISPLAY.SetFocus();
        }
      }
      else
      {
        ptrDlg->activeProccess = FALSE;
      }
    }

    // Check signal controlling to send data.
    if (ptrDlg->sendActivate && (ptrDlg->m_monitorSend.GetLength() > 0))
    {
      unsigned long len;
      SCC::serialCtl().write_scc(ptrDlg->m_monitorSend ,
        ptrDlg->m_monitorSend.GetLength(),len);
      ptrDlg->sendActivate = false;
      ptrDlg->SetDlgItemText(IDC_MONITOR_SEND,"");
    }

    // Check status and signal controlling to close serial communication.
    if (ptrDlg->closePortActivate)
    {
      if (SCC::serialCtl().closePort() == TRUE)
      {
        // Show message that close when performming of closing port okay.
        ptrDlg->SetDlgItemText(IDC_STATUS_PORT,"Closed");
        ptrDlg->closePortActivate = false;
      }
    }
  }
}
return 0;



OL SerialThread::ByteArrayToHexString(PBYTE pbuff, long length, LPTSTR outgoing)

  int i;
  LPBYTE pbuff2;
  pbuff2 = new BYTE[length*2+1];//2 characters per byte, plus one extra for the null terminator

  LPTSTR outgoing1;

  for(i=0;i<length;i++)
  {
      pbuff2[i*2] = UpperNibbleToChar(pbuff[i]);
      pbuff2[i*2+1] = LowerNibbleToChar(pbuff[i]);
  }
  pbuff2[length*2]=0; //null terminate the string

  //This constructor uses SysAllocString so free pbuff2 when we are done
  outgoing1 = (LPTSTR) pbuff2;//(char*)pbuff2;

  memcpy(outgoing, outgoing1 , strlen(outgoing1)+1 ) ;
  delete [] pbuff2 ;
  return 1 ;


ar SerialThread::UpperNibbleToChar(char ch)

      return NibbleToHexChar(ch >> 4);
```

```
ıar SerialThread::LowerNibbleToChar(char ch)

       return NibbleToHexChar(ch & 0xF);


ıar SerialThread::NibbleToHexChar(char ch)

   char hexVal;
   ch = ch & 0xF;
   hexVal = '0'+ch; //for 0 to 9

   if(ch>=0xa)
   {
       hexVal = 'A' - 10 + ch; //set A-F
   }
   return hexVal;


**************************End of file*************************************/
```

```
*********************************************************************

filename:   D:\Applicaiton testing\SerialApp\serialCtl\SerialCtl.hpp
file path:  D:\Applicaiton testing\SerialApp\serialCtl
file base:  SerialCtl
file ext: hpp
author:   Dimitri Denamany.

purpose:  Use to control the serial communication's signal.
*********************************************************************/

fndef SERIAL_CTL_HPP
efine SERIAL_CTL_HPP

 class SerialCtl
---------------------------------------------------------------------------
 Description: This class handle the functionality that interface with
 the serial communication.

ass SerialCtl

blic:
SerialCtl();  // Constructor
~SerialCtl(); // Destructor

blic:
  void setStatusPort(BOOL on_off);     // set Status port whether no or off.
  BOOL closePort();                    // close port operator.


BOOL openPort(DCB dcb,                // open serial communicaiton port.
  const char* portName = "COM1");     // Default port is COM1.

BOOL read_scc(char* inputData,        // read data from serial communication.
  const unsigned int& sizeBuffer,     // sizeBuffer is the size of pakcet that
  unsigned long& length);             // receive from serail port.

BOOL write_scc(LPCVOID data,     // write data to serial communication
  const unsigned int& sizeBuffer,     // sizeBufer is the size of packet that
  unsigned long& length);             // want to send to serial port.

HANDLE getHandlePort();               // The Entry point to get port's handle.
BOOL getStatusPort();                 // The entyy point to get port's staus.

ivate:
BOOL statusPort_;                     // port's status.
  HANDLE handlePort_;                   // the object that is a instace of port.
DCB config_;                          // configuation of serial communication.


ndif //SERIAL_CTL_HPP

**************************End of file*************************************/
```