2006-03-20

Encryption of Computer Peripheral Devices

Kelly Robert Norman
Brigham Young University - Provo

ENCRYPTION OF COMPUTER PERIPHERAL DEVICES

By

Kelly R. Norman

A thesis submitted to the faculty of

Brigham Young University

in partial fulfillment of the requirements for the degree of

Master of Science

School of Technology

Brigham Young University

April 2006

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

Of a thesis submitted by

Kelly R. Norman

This thesis has been read by each member of the following graduate committee and by majority vote has been found to be satisfactory.

Date	Michael G. Bailey, Chair
Date	C. Richard G. Helps
Date	Joseph J. Ekstrom

BRIGHAM YOUNG UNIVERSITY

As chair of the candidate's graduate committee, I have read the thesis of Kelly R, Norman in its final form and have found that (1) its format, citations, and bibliographical style are consistent and acceptable and fulfill university and department style requirements; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the graduate committee and is ready for submission to the university library.

_____	_____
Date	Michael G. Bailey Chair, Graduate Committee

Approved for the Department

_____	_____
Date	Val D. Hawks Graduate Coordinator, School of Technology

Accepted for the College

_____	_____
Date	Alan Parkinson Dean, Ira A. Fulton College of Engineering and Technology

ABSTRACT

ENCRYPTION OF COMPUTER PERIPHERAL DEVICES

Kelly R. Norman

School of Technology

Master of Science

Computer peripherals, such as keyboards, scanners, printers, cameras, and Personal Data Assistants (PDAs) typically communicate with a host PC via an unencrypted protocol, leaving them vulnerable to eavesdropping techniques, such as keyloggers.

An encryption system was developed that is simple enough to be used in peripherals that do not have large amounts of processing power and memory. A software driver loaded in the operating system of the host computer communicates with a simple 8-bit microcontroller in the peripheral device. The driver handles key generation, key exchange, and provides decrypted data to the operating system. A key exchange protocol allows the driver and microcontroller to securely exchange randomly generated keys. The system can function without user intervention, but will alert a user if a non-encrypting or non-authorized peripheral device is detected.

The system is designed to be implemented over a variety of interfaces including PS/2, RS-232, TCP/IP over Ethernet, 802.11, and Bluetooth. A demonstration system was built, which encrypts data on the PS/2 bus between a keyboard and the host computer. Several ciphers were considered for use in encryption. The RC4 cipher was selected for encrypting and decrypting the data in a demonstration system because of it's speed and efficiency when working with 8-bit data. The driver and the microcontroller share a hard-coded key, which is used to encrypt a randomly generated session key, in order to provide a secure exchange of the session key.

The demonstration system performs well, without introducing enough latency to be noticed by the user, and the microcontroller is idle over 95% of the time, even when a fast typist is using the keyboard.

ACKNOWLEDGMENTS

I would like to thank Dr. Michael Bailey for guiding me and helping me stay on track during this long process. His suggestions, contributions, and patience have been a invaluable during the course of my studies. I would also like to thank Richard Helps and Joseph Ekstrom for their support and help as members of my committee, along with all of the BYU faculty who have given me instruction and advice during my undergraduate and graduate studies. The staff of the School of Technology, particularly Ruth Ann Lowe, have been a great help as well.

I owe a tremendous debt of gratitude to my wife, Karlie, for giving me so much support for so long. Her love and unending patience have been some of the greatest blessings I have. She has done at least as much work for this thesis as I have, and she deserves much of the credit. I also want to thank my two children for being so loving, good, and inspirational to me.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background

Data security is a serious matter, and has become an increasingly important issue since the Internet has become a part of so many aspects of everyday life. E-commerce and other activities that require submission of personal and confidential data across the Internet rely on encryption and other forms of protection from the theft of private information. This encryption is focused primarily on protecting information that travels across the Internet, and the information that is stored on a computer. Computer peripherals, however present a point of attack that may not be as strongly protected.

For example, data thieves can purchase, for around $90, a device called a hardware keylogger that can record up to 128,000 keystrokes, can be quickly, easily, and secretly installed on any computer using a PS/2 keyboard, and can be used to steal all information entered through the keyboard, such as usernames, passwords, PIN numbers, credit card numbers, Social Security Numbers, and other confidential information. More expensive models have a capacity of up to 2 million keystrokes. USB keystroke loggers are also available for USB keyboards and peripherals. Public computer labs, such as those on college campuses or public libraries can be a target for anyone wanting to learn people's online banking passwords, credit card numbers, or e-mail passwords. These

devices can also be used in corporate and government environments, to steal confidential trade secrets, military secrets, and other information. Other interfaces are also vulnerable: according to a PC World article (Brandt, 2003), RF keyboards made by HP have been known to transmit key presses to the wrong computer.

Other types of computer peripherals (and other interfaces besides PS/2 and USB) are vulnerable as well. Data transmitted between a personal computer and a printer, scanner, PDA, or camera is potentially vulnerable, whether it is transmitted over a wired interface such as PS/2, RS-232, USB, FireWire (IEEE 1394), Ethernet, or over a wireless interface such as 802.11*x*, Bluetooth, Zigbee, IRDA, or a proprietary protocol.

One type of device for which secure data transmission is very important is biometric security devices. Thumbprint readers, iris scanners, and other devices used to verify a person's identity send identifying information to a computer for analysis. If this information is not encrypted, it can potentially be read directly from the transmission medium in the same manner as keystrokes from a keyboard. The information can then be recorded and used by unauthorized persons.

This research has for its goal to produce a system that may be used in a wide variety of devices, using a variety of media and protocols, to encrypt data as it is sent between a peripheral and a personal computer. The system will use a cipher which can be easily programmed into the firmware of a peripheral device. The decryption will take place in the device driver, in the computer's operating system, so that the application that is using the incoming data will be able to access the decrypted data with no knowledge of the encryption. The data would be encrypted at a byte level, using a fast and simple

algorithm, in order to universally accommodate the simple processors used in most peripheral devices.

The system will be demonstrated by a microcontroller attached to the PS/2 output of a standard PC keyboard. The microcontroller will, after the keyboard has powered up, negotiate a key exchange with the keyboard driver on the host computer. The microcontroller will then intercept all keystroke data sent from the keyboard and encrypt it. The software driver will decrypt the keystroke data and pass it on to the operating system as a normal keystroke.

Ideas that the research will focus on include the following questions:

1) What types of encryption algorithms will be suitable (in terms of memory and processor time usage)?
2) How can the system be modularized to allow for interchanging of ciphers?
3) What is an efficient system for generating and exchanging session keys?
4) How secure can the encryption be using a simple 8-bit microcontroller?

This research will focus on encrypting data between PS/2 keyboards on Intel x86-compatible PCs running a distribution of the GNU/Linux operating system. The research will not cover MS Windows, Macintosh, or other operating systems, or other hardware architectures or keyboard connection types. Although this research may affect the functionality of certain types of key logging software, it is not intended to detect or protect against it. (Administrators of computer labs should be taking steps to restrict installation of unapproved software). It will not focus on detecting or eliminating hardware keystroke loggers, but will instead focus on making them useless by encrypting data so that only the host computer can decipher it. This research is meant only to protect

against reading of data directly from the medium that the computer shares with the peripheral device. It does not provide protection from acoustic analysis of keystroke recordings or from visual monitoring of computer use. It is also not meant to protect against cameras that monitor keyboards or acoustical analysis of key press sounds.

The assumption is made that a would-be data thief will not attempt to open the housing of the peripheral to probe the input to the encryption microcontroller for unencrypted data. This would most likely not be an issue with a production system, as the encryption functionality would most likely be integrated into the existing microcontrollers of peripheral devices. All unencrypted data would strictly be internal to the microcontroller.

1.2 Methodology

The encryption system comprises two separate parts:

1) the microcontroller embedded in the keyboard

2) the software driver, installed in the operating system

The microcontroller's primary task is to receive data from the keyboard, encrypt it using a cipher and key common to both the driver and itself, and to transmit the encrypted data on the PS/2 bus to the motherboard. The driver's primary task is to receive the encrypted data from the BIOS of the system's motherboard, decrypt it, and send it along to the operating system for whatever purpose it was intended.

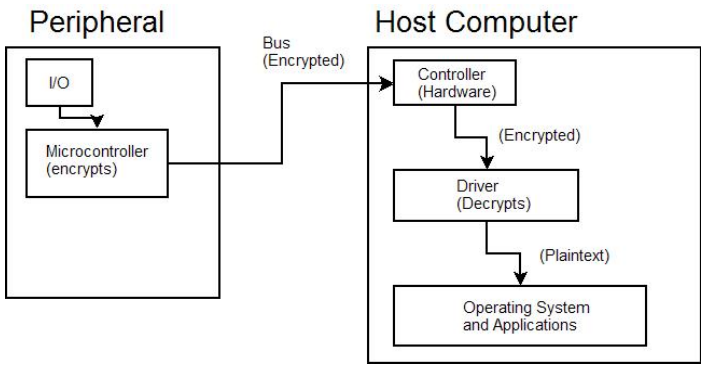

Figure 1-1: Block Diagram of Encryption System

Subtasks involved with the microcontroller include

1) Interfacing with the PS/2 bus on both the keyboard and motherboard sides (receiving data from the keyboard, and sending and receiving data to and from the motherboard),

2) Generating data and passing it to the driver in order to generate new keys, and performing the actual encryption.

3) The driver will need to be able to interface with both the BIOS and the operating system to receive encrypted data, decode it, and pass it on to the operating system, and generate data, passing it to the microcontroller for the actual creation of a key.

The system will operate as follows: when the computer powers up initially, the microcontroller sends data unencrypted so that the BIOS can still interpret key presses correctly. When the operating system starts and the driver is loaded, an initialization command is sent to the microcontroller, indicating that from now on, all key presses will need to be encrypted. The operating system's software driver for the keyboard will, upon resetting the keyboard and receiving the Basic Assurance Test (BAT) code, randomly generate a session key, encrypt it using a unique common key which is hard-coded into

both the driver and the microcontroller, and transmit it to the microcontroller. The microcontroller will decrypt the encrypted session key as it receives it and use it to initialize the cipher. The driver will also initialize its cipher with the same key, and all communication from the keyboard to the driver will be encrypted using the session key from that point forward.

The microcontroller must be selected according to availability, power consumption, available coding/debugging tools, processing power, and ease of programming. Power consumption is an important issue because the microcontroller must be powered by the PS/2 bus, with enough power left over to run the keyboard itself, as well as a PS/2 mouse, if there is one being used on a particular system. Data width is not a critical issue because keyboard data on the PS/2 bus is only 8 bits wide, which most modern microcontrollers can easily handle. More processing power and/or library functions may be required for accomplishing computations relating to the actual encryption. Clock speed is not expected to be an issue, provided that a cipher is chosen which can work well on a simple microcontroller. Typing speeds are very slow compared with the speeds at which most microcontrollers operate, which means that as each key is pressed, the microcontroller should be capable of encrypting the data and sending it, with ample time left over before the next key press. The typematic rate of a keyboard (the rate at which a scan code is repeated if the key is held down) has a maximum value of 30 repeats per second (33 ms between scan codes) (Chapweske, 2003). Fast typists do not approach this rate. If the microcontroller can encrypt and retransmit each scan code received within half of the time between scan codes, it can be considered successful.

Development of the driver will be done with the use of a C compiler such as gcc, which is available as part of the GNU project, and is widely available. Software engineering principles will be important to make the driver in the host computer run efficiently, take up little space in memory, and use variables efficiently and safely.

Several different encryption ciphers will be examined. The criteria for a successful system are that it uses a cipher that is accepted as secure in the cryptography community, in a secure implementation, and that it does not adversely affect system performance.

Chapter 2

Review of Literature

2.1 Establishment of the Problem

2.1.1 Human-Computer Interaction

The devices that allow a person to interact with a computer are called computer peripherals. Computer peripherals typically allow a user to enter information into a computer, retrieve information from it, or both. With the tremendous growth of the Internet in the past decade, millions of people and organizations from average home personal computer users to enterprise-level businesses, find themselves using computers to store, process and transmit data that is sensitive and should be kept private. Usernames and passwords, bank account and PIN numbers, biometric authentication data, financial information, trade secrets, and other information must be protected. Normally, computer systems provide ways of guarding data while it is being stored or transmitted from one computer to another, but keeping data safe while it is being entered into the computer from a peripheral is a separate problem which is not usually addressed as thoroughly.

2.1.2 PS/2 Keyloggers

Hardware-based keystroke recorders are small devices that are connected between a keyboard and a computer's motherboard (Treat, 2002). Typically the connection is made right at the motherboard's PS/2 port, where the device will be out of sight during

normal operation. It records each key press, allowing its owner to remove it and recover the data at his or her leisure. A hardware keystroke recorder can be smaller than an adult's little finger, making it easy to quickly install when nobody is looking, and difficult to detect once it is in place.

When a keylogger is connected to a computer, it is inserted between the keyboard and the host computer. Every byte of data that the keyboard sends to the host must be received by the keylogger as if the keylogger is the host. The keylogger stores each of these bytes in its flash memory, and transmits each byte to the host computer, as if the keylogger was the keyboard.

When the owner of the keylogger wants to view the logged keystrokes, he or she will log in to the system, open a text editor (such as notepad.exe in MS Windows) and enter a predefined password. The keylogger will monitor the sequence of incoming bytes from the keyboard to see if the bytes correspond with the password. If the complete password is received, it will send scan codes to the host computer to display a menu within the text editor. The keylogger owner can choose from options such as changing the password, clearing the memory, or displaying all of the saved keystrokes. Displaying the saved keystrokes causes the keylogger to retransmit all of the keystrokes saved in its memory, effectively "retyping" them into the open text editor, where all of the typed characters can be viewed and saved to a file if desired.

Using www.google.com to search the Internet for the phrase "spy keylogger" returns 32,900 different websites, many of which are selling products that record what people type on their computers. One of these websites, http://www.keyloggers.com/bigbrother.html lists 13 different hardware keystroke

recorders (Raytown Corp, 2005), which range in price from $59 to $299, and can record anywhere from 32,000 to over 2,000,000 keystrokes.

The companies that sell this equipment can often be found by doing Internet searches for terms like "spy" and "espionage," but their websites typically (but not always) include a legal disclaimer explaining that the only legitimate uses for the devices are monitoring employees, spouses, and children to detect inappropriate Internet usage, and that all users of the monitored computer must be made aware (Allen Concepts, 2003). Some manufacturers, such as Allen Concepts, even say that each device has a serial number that can be used to track the purchaser in the event that the device is found to be used illegally. Robert S. Mueller, who was then the Assistant Attorney General in the Criminal Division of the Department of Justice, advised in a letter (Mueller, 1992) that system administrators use banners to alert computer users if the computer they are using is being monitored.

By far the most well-known case involving keystroke recorders has been that of the United States v. Scarfo (McCullagh, 2001). In this case, the FBI used some type of keystroke recorder (whether the recorder was hardware or software-based was never disclosed) to discover the PGP password of suspected loan shark Nicodemo S. Scarfo. The defense claimed that the recorder was used illegally, and attempted to have the FBI forced into giving the details of the technology they used. The Prosecution claimed that divulging this information would put "national security" at risk, and would not publicly disclose it. Ultimately, the evidence gained from the recorder was upheld, and Scarfo was convicted.

Many people have felt that Scarfo's constitutional rights were violated by the FBI's use of the recorder, because the recorder monitored all keystrokes, and not merely those that pertained to the investigation. This may or may not be true, but the legality of that investigation is not the focus of this work. The issue addressed here is that while the FBI's technology may have been sensitive at the time, there are products that accomplish the same task which are readily available to anyone in the world, creating a potential danger for legitimate computer users, especially those who use computers in public areas, where any person off the street could have physical access to the computer's keyboard port. Unless he or she looks at the back of the computer, (which in a public setting is usually installed into some type of cabinet) a user or a system administrator would never even know the device was there.

Universities and public libraries, as well as other places with open computer labs would benefit from a solution to this potential problem, protecting their patrons from identity theft. All companies have sensitive records on employees, customers, new technologies, or other subjects that are best kept private. Various government agencies have classified information that is meant to stay secret. The easy availability of logging devices combined with fairly easy physical access to most computers represents a significant security risk for many computer users.

2.2 Peripheral Interfaces

2.2.1 PS/2 Technical Details

PS/2 is a point-to-point link between the peripheral and the host computer, which means that the host and the peripheral are the only two devices that communicate on that

link. In order for a sniffer device to listen to data sent over a PS/2 connection, it would need to mimic the functionality of the host computer to the peripheral device and mimic the functionality of the peripheral to the host computer, passing data back and forth as it records it. Other point-to-point interfaces include IEEE 1284 (parallel printer ports) and RS-232 (serial ports). Other interfaces, such as Ethernet, and Universal Serial Bus (USB), use a bus topology where each device on the bus can listen to and communicate with each of the other devices on the bus. On such a bus, a sniffer device can easily listen for data without having to be placed in between and mimicking the host and the peripheral device.

A USB version of a keystroke logging device is also available from keykatcher.com. USB has become a very popular interface for connecting almost any type of computer peripheral to a host computer. Keyboards, mice, scanners, still and video cameras, removable storage devices and biometric authentication devices such as fingerprint or retinal scanners use USB, so in theory, a sniffing device could be designed to listen to nearly any other type of peripheral on the Universal Serial Bus in the same way that the keyboard sniffer does. These devices could all benefit by being protected by encryption (Treat, 2002).

2.2.2 USB Technical Details

USB uses a number of different driver layers. It also has a system of message pipes for handling different types of control and communications with any one device. A pipe is a logical connection between the driver and the device, similar to a socket in TCP/IP. The driver can create a control pipe or a message pipe to communicate with a

specific part of the device. In a USB interface the software driver would be at the USB Driver level (the interface between the USB System Software and the client software). The USB Driver can create a control pipe to handle all of the communication for the encrypted key exchange and synchronization. Once the cipher has been initialized with the session key a new message pipe can be created where all data in both directions is encrypted (USB Implementers Forum, Inc., 2000). Data from the client application is then transmitted over this pipe.

2.2.3 RS-232 Technical Details

RS-232 is an interface that allows two devices to communicate over a point-to-point link (similar to PS/2). RS-232 has no specific protocol defined for it, so each device that uses it is designed with its own protocol. An RS-232 connection has a ground wire, a transmit (TX) wire, and a receive (RX) wire, in addition to other wires used for control purposes. Different devices make more or less use of these other wires. Many devices only use TX, RX and Ground, and some only communicate in one direction, so they would not need either the TX or RX wire (Strangio, 2006).

An application that communicates via an RS-232 interface sends to and receives from the RS-232 port using the port's driver (COM1, COM2, etc. on an MS Windows system, or ttyS1, ttyS2, etc. on a GNU/Linux system), as opposed to using a driver specifically for the device. The driver for the port would handle the encryption, and would initiate encryption when the system is powered on, or on receiving the first byte of data from the peripheral connected to that port.

2.2.4 TCP/IP over Ethernet Technical Details

TCP/IP is the name of the suite of communications protocols that computers use to communicate via the Internet. TCP/IP allows devices to communicate with each other through across local networks as well as across the Internet. Each device has a unique 4-byte address. Computers on a common network will normally have addresses similar but not identical to each other. Each application uses one or more specific logical ports to communicate. When a device receives a TCP/IP packet it will examine the packet to see which port it is directed to, then send the data from the packet to which ever application is using that port (Crowfroft et al, 2002). In TCP/IP applications, port numbers are chosen when the application is designed. Many applications will allow a server to be configured to use a port other than the default, but certain port numbers are reserved for some of the more common applications. For example, port number 22 is used for ssh (Secure Shell), port number 23 is used for telnet (a terminal application), and port 80 is the default port for http (Hypertext Transport Protocol, or web) servers.

Hubs and switches are devices which allow computers to connect to a common network. Routers allow different networks to be connected to each other. TCP/IP provides the mechanism for routers to determine which network to send any given packet to.

TCP/IP operates on multiple layers. These layers can be conceptualized in different ways, but they can be thought of as the Application, Transport, Network, Driver, and Physical (wire) layers. TCP, and other protocols in the TCP/IP suite, are located in the Transport layer of this stack. TCP handles connections from end to end, maintaining an active session between two communicating devices. The IP protocol handles the

Network layer of the stack, routing packets across and within networks. IP does not deal with connections.

When an application has data to send, it passes the data and the destination address via an application programming interface (API) to the transport protocol layer for the correct protocol (such as UDP, TCP, etc). The kernel passes the data to the IP level, which adds the necessary IP protocol information, then puts it in a queue for output. The device then transmits the data (Crowfroft et al, 2002). This process is illustrated in Figure 2-2.

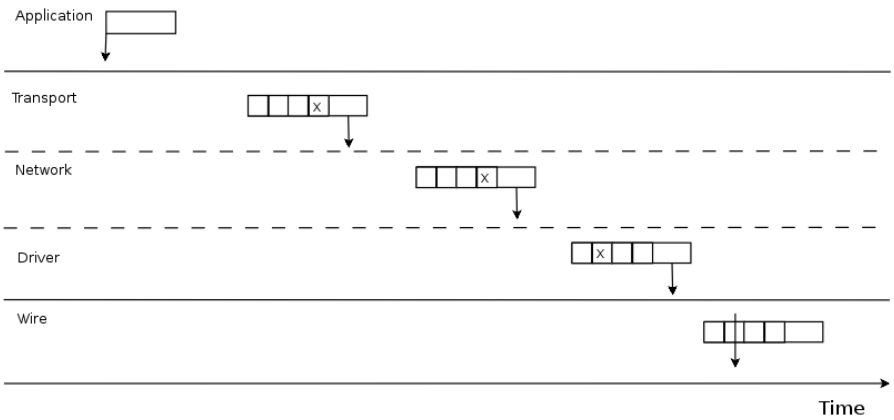

Figure 2-1: TCP/IP Packet Transmission across Network Layers (Crowfroft et al, 2002)

The X in each layer represents the data from the previous layer. Each layer adds its own relevant data to the packet as the packet passes down through the stack. This added data is represented in Figure 2-2 by the boxes surrounding the X box. Each layer adds its own addressing information, as well as checksums and other mechanisms to ensure the integrity of the message.

When data is received, the data passes through the same layers in the reverse order. The data is taken from the network device and placed in memory. The driver is

alerted that the data is ready by the general system scheduler. The IP protocol layer checks the packets and determines whether the packet should go to a local application or if it should be forwarded elsewhere. The IP layer determines which transport protocol should handle the data (TCP, UDP, etc.) and passes it to the correct one. The transport protocol checks the packet header and passes the data to the correct socket. Once the data is queued for the correct process, the process waiting for the data is awoken so that it can receive the data and process it.

TCP/IP is typically used for communications between computers, not normally for computer peripherals. Some types of peripherals, such as Personal Data Assistants (PDAs) can use TCP/IP to communicate with a host computer. There are already several common methods for encrypting TCP/IP traffic, including Secure Sockets Layer (SSL) and Secure Shell (SSH). These protocols are already established as standards, and widely used for Internet/network traffic. Open source implementations of SSL and SSH exist and are popular (OpenSSH, OpenSSL). Any peripheral devices that require secure communication over a TCP/IP connection can use SSL, SSH, the free implementations, or another secure protocol. Because of the limited use of TCP/IP for peripheral communication, as well as the existing encryption protocols for TCP/IP, there is little need for a new TCP/IP peripheral encryption system.

2.2.5 Wireless Interfaces

Wireless devices using both radio frequency and infrared technologies are even more susceptible to snooping, because data recording devices do not require a physical connection. In some cases, they do not even require close proximity. PDA's that

synchronize with PCs with IR or RF links, remote controls that could be used to enter passwords, the new Smart Displays being developed by Microsoft and others could use protection from snoopers in the form of encryption. In one instance, a wireless (RF) keyboard used by a Norwegian man in his own apartment was transmitting his keystrokes to a neighbor's computer over 100 yards away, without the keyboard owner's knowledge (Brandt, 2003). If an incident like that can happen by accident, it can be reproduced intentionally for the purpose of stealing information.

2.2.5.1 802.11 (Wi-Fi)

The wireless networking standard 802.11 provides options for security, but those options often go unused, and the most common of them do not provide adequate security. In July 2002, a hacker purchased a wireless networking card for his laptop at a Best Buy store, installed it in his laptop in the store's parking lot, then used it to access the store's wireless network. Through the company's wireless network, he was able to access sales data, which included customers' credit card numbers and other personal and financial information (Bolles, 2002). "Wardriving" is the act of discovering wireless networks while driving in a car, using a computer with a wireless network adapter. Some wardrivers only search for wireless access points that do not use encryption ("open" access points), and log their findings, sometimes on public internet sites. Others attempt to use open access points for their own Internet use, and still others attempt to break the encryption used in many access points, either to use the access point owners' Internet connection, or to hack into the computers on the networks (Lawrence et al, 2004). The lack of security on many wireless networks indicates that any device that communicates

across an 802.11 network, whether a peripheral or a host computer, must be designed with security (including encryption) in mind.

802.11a/b/g has an optional security component called Wireless Encryption Protocol (WEP). WEP uses the RC4 stream cipher to encrypt data. The implementation of RC4 in WEP, however, has been found to have serious weaknesses (Wong, 2003) (Fluhrer, 2001). Despite the fact that WEP-protected network access points have options to not broadcast the network's name and to use keys of varying lengths, WEP-protected access points are often quickly and easily compromised. Newer security protocols such as Wi-Fi Protected Access (WPA) and 802.11i have been designed, which provide stronger encryption, but there is always a possibility that whatever encryption scheme is used by a wireless network, weaknesses may easily be found, or however advanced and secure the encryption may be, many users will still not turn it on or configure it properly. Peripheral devices which communicate over a wireless network should have a layer of encryption beyond what may or may not be offered on any given network.

Wi-Fi uses the same protocols that are normally supported on an Ethernet TCP/IP network, but embeds the TCP/IP packets in additional layers that contain information such as the name of the wireless network, and the MAC address of the wireless network adapter.

2.2.5.2 Bluetooth

Bluetooth is a networking protocol designed to allow devices to create small networks with a limited range (Bluetooth SIG, 2006). Bluetooth allows keyboards and mice to communicate with a host computer, and it also allows devices such as PDAs

(Personal Digital Assistants) and cell phones to transfer address books, calendars, and other files. Bluetooth devices discover each other and configure themselves automatically without requiring user intervention. A user can be sitting in a public place like a bus, or a park bench, and have his or her laptop computer connect via Bluetooth to his or her cell phone, and automatically transfer data. But in public areas, other people can exploit weaknesses in the communications protocol. Bluetooth has had its share of security vulnerabilities (Shaked et al, 2005) (Hager et al, 2003) (Potter, 2005). These vulnerabilities range from problems with how Bluetooth application software is designed, to methods to discover a user's 4 digit PIN, which allows the user to control which devices are allowed to connect to any one given device. Bluetooth devices would benefit from having a well-designed security layer beyond what the Bluetooth protocol specifies.

2.2.5.3 Bluetooth Technical Details

Bluetooth supports a number of different protocols for communication, including OBEX (Object Exchange), TCP, RFCOMM (designed to emulate RS-232 connections), and others (Huang, 2005, p. 20 – 23). Any approach for encryption over a wired Ethernet TCP/IP link stated above would function for a Bluetooth TCP/IP connection as well. A more comprehensive approach could possibly be more useful, because each of the protocols could be protected by one common encryption system.

Bluetooth protocols have a limited number of ports (or protocol service multiplexers, as they are typically referred to in Bluetooth terminology), so instead of arbitrarily choosing port number for each application at design time, ports are chosen when two devices connect to each other by means of the Service Discovery Protocol

(SDP) (Huang, 2005). The SDP allows devices to connect to each other using a reserved port for the L2CAP protocol. Applications are dynamically assigned port numbers at runtime, and these port numbers are registered with the SDP server, which also registers a description of the services offered by each application. When a client connects to the SDP server, the client selects the correct service description, and the SDP server sends the corresponding port number to the client.

2.3 Existing Solutions

Typically, computer users are left to their own vigilance to avoid privacy breaches, whether using computers in a public lab, personal devices such as a PDA in public, or using a home computer.

The type of information that is stored on and used by a computer system, as well as the purpose and location of the computer, should determine the level of security the computer uses (Webb, 2004). A typical home computer user will not normally employ extensive security measures, because there are not normally a large amount of people who have access to the computer. A computer in a public lab requires greater security because many people share the same computer. Computer lab administrators may restrict usage of lab computers to a certain group of people, such as students currently enrolled in classes, or library card holders. A computer lab will also normally enforce policies that keep users from changing system settings. A computer used in top secret military installations will normally be not connected to the global Internet, and restrictions will be placed on who is allowed to enter the computer room.

2.3.1 Physical Access

The first step to guarding private data is to make sure that only people who need to use a computer are able to use the computer. If a computer is used for very sensitive purposes (e.g., financial data, secret military data, or corporate trade secrets), physical access to the computer should be restricted. If unauthorized users are not allowed in the room with the computer, they will not be able to make hardware modifications to the system, or be able to physically remove data storage devices. Also, depending on the application, access to the Internet or other internal networks should be cut off or restricted to prevent the possibility of hackers monitoring data transfers over the network, and to ensure that they cannot exploit security flaws in the computer's operating system or applications to remotely gain control of the system. In classrooms, school computer labs, and libraries, the computer case may be kept in a locked cabinet, so that the computer is available for people to use, but there is still no physical access to the machine.

As mentioned previously, wireless networking provides another avenue of physical access for malefactors who are within the physical range of a wireless network, even if they do not have direct physical access to the computers themselves. Even properly configured wireless networks can be simple to break into with the correct tools. The best defense against unauthorized wireless network access is to avoid using a wireless network. If a wireless network must be used, WPA2 security should be used, as WEP encryption is very simple to break.

2.3.2 User Access Control (Software)

Any computer that deals with sensitive data should have an operating system that restricts users according to their status. Administrators have full access to data and system settings. Normal users have access to their own data and settings that affect their own computing sessions (color schemes, fonts, etc.). Guest users, if allowed, have access to very little settings or data, if any at all.

User authentication is done by means of a login screen at the beginning of a session. If the computer is unused for any significant amount of time, the screen should blank or switch to a screensaver to prevent passersby from being able to see data on the monitor, and the user's password should be required to exit the screensaver.

2.3.3 Encryption

Encryption is a way to obscure data so that it is very difficult for the data to be correctly understood by any unauthorized persons. Data can be encrypted when it is held on a non-volatile storage device, such as a hard disk drive, optical disc or a flash-based memory card. It can also be encrypted when it is transmitted from one device to another. There are two main categories of encryption: symmetric key, and public key (Stallings, 2002). Both types of encryption use complex algorithms to change the values of the data in a way that is reversible only for someone who knows the correct key.

2.3.3.1 Symmetric Key Encryption

Symmetric key encryption is a category of encryption algorithms where the data is encrypted and decrypted using the same key. Key lengths in symmetric encryption can

be from 64 bits to 256 or more bits. Symmetric key ciphers normally use simple operations that can be easily done general purpose processors (XOR, bit shifts, substitutions), but the operations are arranged in a complex order (Stallings, 2002). Symmetric key ciphers include AES (Advanced Encryption Standard), DES (Data Encryption Standard), and Blowfish (Stallings, 2002) (Schneier, 1993).

One disadvantage to symmetric key encryption is that if two parties, Alice and Bob, need to exchange encrypted information, they need to have the same key. This implies that they need a secure way to agree on a key to begin with. If Alice chooses the key, then she must somehow transmit it to Bob. Since the key is critical to the security of the encrypted data, Alice needs a very secure method of getting to key to Bob. If Alice and Bob are in different locations and can only communicate electronically, they will most likely need to encrypt the key to make it secure. A good way to handle encryption for the key exchange is to use public key encryption.

Symmetric key ciphers fall into two categories: block ciphers and stream ciphers.

2.3.3.1.1 Block Ciphers

Block ciphers operate on a group of bytes all at once. Most block ciphers have a block size between 32 and 256 bits. In an ideal block cipher, a change in any one of the data bits in a given block can affect each of the other bits, which means that two similar blocks of plaintext would look completely different from each other after encryption. Block ciphers use a complicated set of operations, but the operations are done on an entire block at once, so they are reasonably efficient as long as there is enough data to fill at least one block. If a given set of data does not fill a complete block, the remaining bits

of the block are typically zero-padded to give the proper block length. AES, DES, Blowfish, and RC5 are examples of block ciphers (Stallings, 2002).

2.3.3.1.1.1 AES

AES (Advanced Encryption Standard) was developed as a replacement to the widely-used DES (Data Encryption Standard) (Stallings, 2002). The National Institute of Standards and Technology accepted 15 algorithms as proposals for the new standard, and eventually chose the Rijndael proposal, based on its level of security, cost, and implementation characteristics.

The Rijndael algorithm is a block cipher that can be used with 128, 192, or 256 bit keys. It has no known attacks, has a relatively simple structure, can easily be implemented in software or hardware on platforms from 8 to 64 bits, and works particularly well in environments with a restricted amount of ROM and/or RAM. For a 128 bit key, 10 rounds are used, with each round comprised of an S-box based substitution stage, a simple permutation, and two more substitution stages. The Rijndael algorithm can be implemented in a Cipher Feedback mode or an Output Feedback mode, allowing it to be used as a stream cipher.

2.3.3.1.1.2 RC5

RC5 is a block cipher which was developed by Ron Rivest of RSA Security. RC5 has a variable block size, number of rounds, and key length. The block size can be 32, 64, or 128 bits. The number of rounds can range from 0 to 255, and the key length can be up to 2040 bits. The variable parameters of RC5 make it possible to lower the processor

requirements, but the strength of the encryption will be reduced correspondingly. Systems with more capable processors and/or more memory can use higher values for each of the parameters to increase the strength of the cipher. RC5 uses addition, bitwise XOR, and circular rotation operations, which are all commonly found on microprocessors (Stallings, 2002).

2.3.3.1.1.3 Blowfish

Blowfish was developed by Bruce Schneier in 1993 (Schneier, 1993). Schneier's intention in developing the Blowfish was to create an alternative to DES (the aging predecessor of AES) which was free from legal encumbrances such as patents and copyrights. Blowfish claims to be very efficient in encryption on large microprocessors (all of the data encryption operations are simple XOR operations and additions on 32-bit words), but it uses a complex initialization phase which is fairly memory and processor intensive, which would be difficult to implement on small processors.

Blowfish is a Feistel-based cipher, which performs a simple function on a block of data 16 times. The size of the data block used is 64 bits, and the key length can be as high as 448 bits. The key expansion part of the cipher converts the key to several sub-key arrays for a total of 4168 bytes. This key expansion would be a hurdle for any processor with 4 kB or less of RAM (Schneier, 1995).

Known attacks against Blowfish include an attack on 3-round Blowfish, which does not extend to the full 16-round version, an attack against a simplified Blowfish where the S-boxes are known, which does not work if 8 or more rounds are used, and

certain weak keys (the odds of getting one of the weak keys are 1 in 2^{14}). No attacks are known to be effective against a full 16-round version of Blowfish (Schneier, 1995).

2.3.3.1.2 Stream Ciphers

Stream ciphers are designed to be used in applications where data is not available in large blocks at a time. They operate on smaller chunks of data using fast operations. The typical stream cipher takes the output from a pseudo-random number generator (PRNG) and XOR's it with the data to be encrypted. The cryptographically strong key used in a stream cipher is the seed for the PRNG. Both the encrypting and decrypting parties must use the same PRNG, and they must stay synchronized. If one party misses a transmission from the other, their PRNG's will no longer be synchronized, and the decryption process will try to decrypt the ciphertext by XORing with the wrong number. All subsequent transmissions will not be able to be correctly decrypted until the PRNG is reset by both parties at the same time, using the same key. RC4 is an example of a stream cipher (Stallings, 2002).

Block ciphers can be used as stream ciphers in what is known as Cipher Feedback mode. Instead of using the block cipher to directly encrypt the plaintext, the block cipher encrypts an initialization vector (which is a block of random data used to seed the pseudorandom number generator created by the Cipher Feedback mode), then uses the encrypted output of the cipher as the input for another iteration of the cipher, as if it were plaintext. At each iteration, the output is used as the input for the next iteration, and each output is XORed with a block of plaintext (Stallings, 2002).

2.3.3.1.2.1 RC4

RC4 is a stream cipher originally developed by Ron Rivest, of RSA Security. Of the ciphers considered, RC4 undoubtedly has the simplest implementation, with byte-oriented operations, and no need for large lookup tables (Stallings, 2003). The encryption of each byte takes between 8 and 16 machine cycles, making software execution very fast. The key length can vary up to 256 bytes. A pseudo-random number stream is generated from various permutations of a vector S, which contains 256 8-bit numbers, where each of the numbers from 0 to 255 is found at some position within the vector (Stallings, 2002).

One protocol that uses RC4 has been known to be particularly weak: WEP, the Wired Equivalent Privacy protocol. The attacks on WEP involve the way the key scheduling algorithm of RC4 is used. Because the WEP protocol generates its keys from a common base key, the output of the PRNG during the first 256 iterations is not random enough. After 256 iterations, however, the PRNG generates numbers with sufficient randomness to be considered relatively secure (Fluhrer et al, 2001). These concerns are addressed by Ron Rivest on the RSA Security website: "The initial key scheduling component of RC4 should for now be routinely amended for new applications to include hashing and/or discarding the first 256 bytes of pseudo-random output. (This has in any case been RSA's routine recommendation.)" (Rivest, 2004) Rivest also points out that other protocols, such as SSL, use RC4 and have not had the security problems that WEP has. The demonstration system discards the first 256 bytes of output from the cipher before using the cipher's output for encryption.

Using a key length greater than or equal to 128 bits, no known attack is practical. One cryptanalysis using a CPLD-based system claims to have "an outstanding price/performance ratio, easily beating other low-cost approaches …" (Kundarewich, P.D. et al, 1999). This system could be expected to crack a 32-bit RC4 encryption in approximately 15 hours. The theoretical expected time to crack a 40-bit encryption was 159 days. A key length of 40 bits was used as a reference because that is the maximum key length that can be exported from the U.S. and Canada. Continuing the mathematical process to 64 bits, the expected time is over 7,354,396 years to crack, and 1.35×10^{26} years for 128 bits.

Advantages:

- Very fast execution (8-16 instruction cycles)
- Very low memory requirements (about 512 bytes, plus the length of the key)
- Stream Cipher which encrypts one byte at a time
- Simple algorithm, easy to implement

Disadvantages:

- Key scheduling can have weaknesses depending on implementation

2.3.3.2 Public Key Encryption

In public key encryption, one party uses a key to encrypt the data, but an entirely different key is used for decryption. The key used for encryption is called the public key, and the key used for decryption is called the private key. One key can not be used to

deduce the other. If two parties, Alice and Bob, want to use public key encryption to communicate, they each have a private key and a public key. The private key is kept secret by each person. The public key may be given out freely to anyone, including parties who are not trusted. When Alice wants to encrypt a message to send to Bob, she first needs to obtain Bob's public key. She uses the public key to encrypt her message. Because of the relationship between the two keys, the encrypted message may now only be decrypted using the private key, which only Bob has (Stallings, 2002).

The application of the keys can be reversed to provide a "digital signature" for a message instead of encryption. If Alice encrypts the message using her private key, anyone can use Alice's public key to decrypt the message. If the decrypted message is anything other than garbled data, the receiver of the message can be sure that Alice's private key was used to encrypt it, which Alice doesn't share, so the receiver has verified that the message was indeed sent by Alice. Digital signatures and encryption can be combined to ensure that only the intended receiver can decrypt the message, and to verify that it was sent by the correct sender. To accomplish this, Alice would use her private key to encrypt the message, then use Bob's public key to encrypt it again. Bob would then use his private key to decrypt the message, then use Alice's public key to decrypt it again (Stallings, 2002).

RSA is an example of public key encryption. RSA uses a product 'n' of two very large prime numbers 'p' and 'q', which is used to create the two keys. Knowing the values of p and q is important to being able to compute the keys. The security of RSA encryption lies in the fact that it is thought to be very difficult to factor n, if p and q are large enough (Stallings, 2002).

2.3.4 Hardware Encryption Solutions

Existing research does not show any unified system for encrypting communications with computer peripherals in general. Some research exists dealing specifically with Smart Cards and a few other specific embedded applications. It is primarily focused on the encryption processors and algorithms used by them for authentication, and for encrypting data for storage on the cards. However, it does not deal with encrypting for transmission to a host computer. There is also some research about the performance of certain ciphers (e.g., AES, RSA) on specific microcontrollers, or microcontrollers designed specifically for one cipher or another, and FPGA implementations of certain ciphers.

2.3.4.1 Software on General Purpose Microcontrollers

Daniel Treat proposed an encryption system for PS/2 keyboards (Treat, 2002). In Treat's system, a microcontroller embedded in the keyboard encrypts data and sends it to the host computer, where it is received by either an additional microcontroller which decrypts it, or the BIOS and PS/2 hardware have the decryption capabilities built in (Treat, 2002). By having the decryption performed by hardware on the host computer's motherboard, this system does not allow use by other peripherals on different interfaces.

Treat's system uses a stream cipher based on a Feistel block as a PRNG. Treat mentions that Feistel ciphers in use today use block sizes far greater than the 8 bits used for each packet in the PS/2 protocol, so a stream cipher would be more appropriate. He then explains that a Feistel cipher may be used as a stream cipher through the Cipher Feedback mode, but concludes that the Cipher Feedback mode carries too much overhead

for use in microcontrollers. His conclusion is that the Feistel cipher itself be used as the PRNG, with a rotating key (the key is incremented after each encryption) to prevent replay attacks (Treat, 2002). This is not a cryptographically strong PRNG

Treat provides a way to create a session key by having the microcontroller/BIOS in the host computer and the keyboard microcontroller generate random numbers (each number being half the length of the initial shared key), exchanging the two random numbers and concatenating them to create an offset, then XORing the offset with the initial shared key (Treat, 2002). He does not provide a way for the offset data to be exchanged securely, nor does he provide a way for a random number to be created by either the keyboard microcontroller or the host computer's decrypting microcontroller.

Xiaohua Luo, Kougen Zheng, Yunhe Pan, and Zhaohui Wu compared several different encryption algorithms for wireless network sensors running on 8-bit AVR microcontrollers (Luo et al, 2004). The ciphers tested were SEAL (a stream cipher with a 160-bit key and 8-bit block size), RC4, RC5 (using a 64-bit key and a 64-bit block size), and TEA (a block cipher with a 128-bit key and 64 bit block size). They measured memory requirements for each cipher, as well as initialization time and encryption and decryption time. All measurements of encryption and decryption times are for processing one 32 byte packet..

SEAL requires more than 4 kB of RAM, which may be a problem for some microcontrollers. RC5 uses nearly 2 kB or memory, which may still be a problem, considering that the microcontroller may have to be able to do work other than just encryption. RC4 and TEA were both well under 1 kB. The initialization time for SEAL was over 5 ms, while RC5 took 2.4 ms to initialize. RC4 initialized in 474 µs, and TEA

took only 10 µs. RC5 was the slowest algorithm for encrypting and decrypting, at roughly 820 µs per 32 byte packet.. SEAL took 213 µs for encryption and decryption. RC4 and TEA were very close in time with 173 µs for RC4 and 161 – 164 µs for TEA (Luo et al, 2004).

Their conclusion is that for 8-bit AVR systems, RC4 is the best choice. RC5 was predicted to do well in 32-bit architectures, but was not recommended for the 8-bit systems. TEA did well in all tests, but attacks on the cipher made it a less desirable choice (Kelsey et al, 1997).

Atasu, Breveglieri, and Macchetti presented an optimized implementation of AES for running on ARM-based processors. Their implementation showed improvements over existing implementations (Atasu et al, 2004).

2.3.4.2 Encryption-Specific Hardware

One way of successfully implementing a strong cipher in a small and limited system is to use specialized hardware. An encryption co-processor built from custom-designed ASIC or FPGA circuits can offload the intensive encryption work from the main microcontroller, giving the main microcontroller more capacity to deal with other tasks. Specialized processors are custom-built and optimized specifically for encryption. Typically, they will be built completely around an engine for one specific cipher, or else they are designed to be able to specifically implement certain operations, or handling data in specific ways that are more useful in encryption, and are focused less on being able to do general-purpose computing operations.

Goodman, Dancy, and Chandrakasan, describe a processor designed specifically for encryption in battery operated wireless devices (Goodman et al, 1998). The processor has an encryption engine based on the Quadratic Residue Cipher (QRC), which is a public-key style stream cipher. QRC generates a pseudo-random stream of numbers by squaring a large number modulo the product of two large prime numbers ($n = p * q$). The result is the first output of the PRNG, which is then squared modulo n to obtain the next pseudo-random number, and so on (Shepherd, 1994). QRC is difficult to implement on small 8-bit microcontrollers because it uses multiple-precision arithmetic. A processor with a hardware-based QRC engine makes it possible to use QRC in an embedded platform. Using multiple-precision arithmetic to decrypt the ciphertext in software on the host computer could be very time consuming. For data with large throughput requirements, this may not be suitable. In applications where an identical processor, or other hardware-based implementation can be used to decrypt the data stream, or in systems with small throughput requirements this would be a viable approach. This QRC-based processor is designed to be power-efficient, and allows the power consumption to be lowered by configuring the unit for less secure encryption (which is accomplished by using a smaller n value) (Goodman et al, 1998).

Chih-Pin Su, Tsung-Fu Lin, Chih-Tsun Huang and Cheng-Wen Wu proposed a design for a processor which performs AES encryption with a high throughput rate (Su et al, 2003). The processor can use key lengths of 128, 192, or 256 bits, and in addition to the actual encryption, the processor implements the key expansion steps of AES. The processor's design was optimized primarily by improving the way the S-Box step of AES is handled. Traditional AES implementations use a lookup table (LUT) to determine the

result of the S-Box step. The S-Box function is actually a multiplicative inverse in GF(2^8), and an "affine transformation (over $GF(2)$): $b(x) = (x7 + x6 + x2 + x) + a(x)(x7 + x6 + x5 + x4 + 1)$ mod $(x8 + 1)$, where $a(x)$ is the multiplicative inverse in polynomial form." (Su et al, 2003) The inverse function can be reduced, using the proper transform function, to finding the inverse over GF(2^4). The processor also implements its 16 S-Boxes in parallel, to be able to process an entire 128-bit block at one time. The architecture of the chip is shown in Figure 2.1.

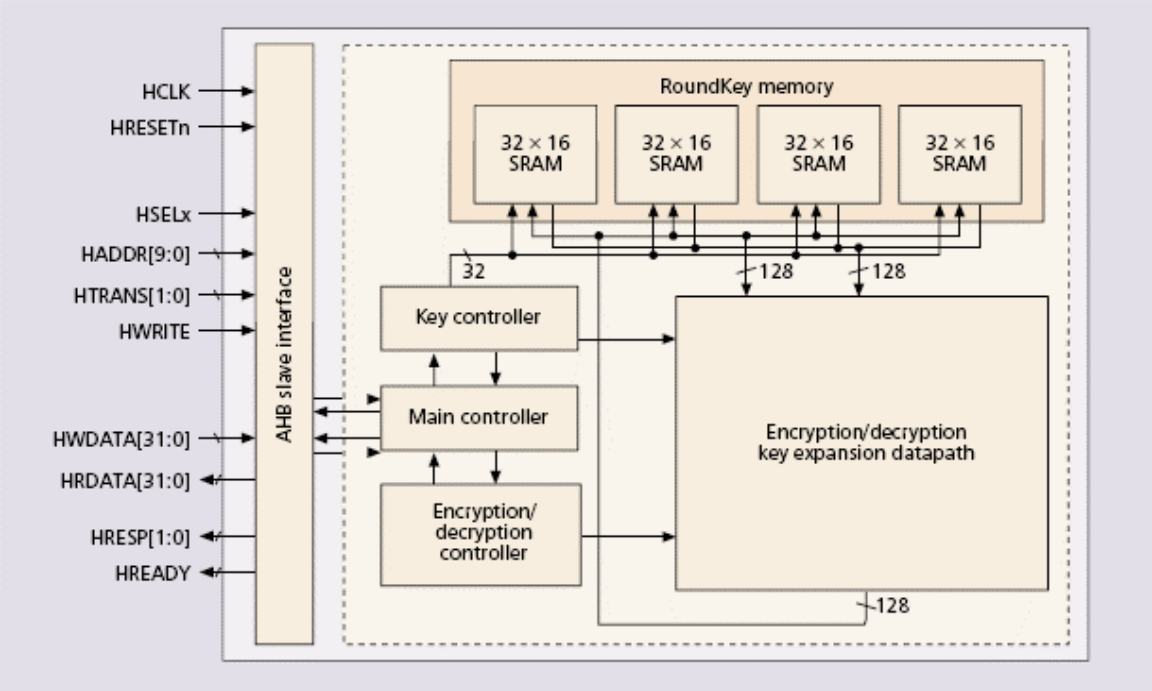

Figure 2-2: AES Encryption Processor Architecture (Su et al, 2003)

This design was able to use higher clock speeds (200 MHz) than most other ASIC-based AES processors that had been proposed at that time (64 MHz – 224.22MHz) to help achieve an increased throughput of 2.008 GB/s (versus .241GB/s – 2.381 for the four other designs it was compared against). The gate count is toward the lower end of

the range of gate counts used in the other designs (58,430 gates, compared to 15,493 – 612,000 gates). This leaves the throughput per gate at 34.98 kb/s/gate. The other designs had throughput per gate measurements of 3.18 kb/s/gate – 122.28 kb/s/gate. (The 122.28 result was the only measurement above 42 kb/s/gate, which makes this design one of the fastest designs out of those designs that were compared.) The one design that scored better in most of the measurements above was designed using .6 μm CMOS technology, as opposed to the 35 μm technology of this design. Also, it was limited to 128-bit keys, whereas this design can implement 128, 192, and 256-bit keys.

2.3.4.3 Trusted Computing

The Trusted Computing Group (TCG) is a not-for-profit organization of companies in the computer and technology industries, which sets standards for hardware-based security systems for computers. The companies involved in the TCG include AMD, Intel, Hewlett-Packard, Microsoft, and Sun Microsystems. According to the TCG website:

> "The Trusted Computing Group (TCG) is a not-for-profit organization formed to develop, define, and promote open standards for hardware-enabled trusted computing and security technologies, including hardware building blocks and software interfaces, across multiple platforms, peripherals, and devices. TCG specifications will enable more secure computing environments without compromising functional integrity, privacy, or individual rights. The primary goal is to help users protect their information assets (data, passwords, keys, etc.) from compromise due to external software attack and physical theft." (http://www.trustedcomputing.org)

One of the main specifications from the TCG is the Trusted Platform Module (TPM). The TPM is a chip that can be integrated into a computer's motherboard, which provides functionality for ensuring that the only approved hardware and software will function with the computer. (Felten, 2003) The TPM has two main purposes: 1) to

provide an authenticated boot, and 2) encryption. The authenticated boot monitors each stage of the system's boot process, in order to be able to provide detailed information to application software about the operating system and other software installed on the computer. The encryption functionality gives the computer a master security key, from which the TPM generates a unique secret key for each possible configuration of the computer. This allows TPM-aware software to know when changes have been made to the computer, such as a new hard drive or processor being installed, or a new operating system being loaded (Felten, 2003). The TPM provides RSA encryption functionality, including tamper-evident storage of the RSA keys. It also provides a random number generator, which can be useful for generating private keys.

The TPM can provide authentication of computer peripherals, so that only approved hardware will be allowed on a system. There does not appear to be an intention to provide encryption for communication between the host computer and the peripheral, but if a TPM-aware software driver can have access to the encryption hardware in the module, it could potentially be used in that capacity.

Trusted Computing is very controversial, as it gives TPM-aware software control over what files can be opened by which applications (Schneier, 2005). A TPM-based solution may not be the best approach for general-purpose (e.g., home use), but may be well-suited to some corporate and military situations, where Trusted Computing would be more important than the freedom to run any software or open any file.

Chapter 3

Methodology

3.1 Solution

The solution being proposed by this work consists of two parts: encryption from within the peripheral device and a software-based peripheral driver that resides in the operating system and performs the decryption. The goal of the work is to find a system that will provide a very secure connection between any type of peripheral and the host computer, while maintaining a natural response time. The microcontroller must be able to encrypt and retransmit data in less than half of the time between the most frequent transmissions. For a keyboard, the maximum data rate is 30 8-bit scan codes per second (33 ms between codes), which means that microcontroller must take less than 16 ms to encrypt and retransmit each byte.

3.2 General Implementation

The overall concept of this encryption system is to allow any type of peripheral device to be able to begin communicating with the host computer, and turn on encryption functionality, automatically and transparently to the user and the application. The process is as follows: (see Fig 3-1)

1) The peripheral device and the host computer each initialize their own cipher with the shared key.
2) The host computer randomly generates a session key, encrypts it using the shared-key cipher, then both devices wait until a connection is made between them.
3) The host computer queries the peripheral for its encryption capabilities.
4) If the peripheral was not designed with encryption capabilities, it naturally ignores the request, or sends some type of error code to indicate it received an unknown command. Otherwise, it sends a data byte with information about its encryption capabilities.
5) On receiving a reply to the encryption query, the host computer sends the encrypted session key to the peripheral.
6) The peripheral re-initializes its cipher with the new session key, then sends a message to the host computer indicating that it is ready to begin encrypting data. From this point, the peripheral encrypts each byte before sending it to the host.
7) The host enables decryption. From this point, every byte the host receives will be decrypted before being passed to the corresponding application.

The response to the query in steps 3 through 5 could be as simple as a Boolean true or false, indicating that the peripheral supports one predetermined encryption cipher. Another possibility could be that the host is equipped to deal with two or more different

ciphers, and the response from the peripheral gives an ID number for which cipher it supports. The host would then use that cipher to encrypt the session key, and then use it to decrypt all incoming data from the peripheral.

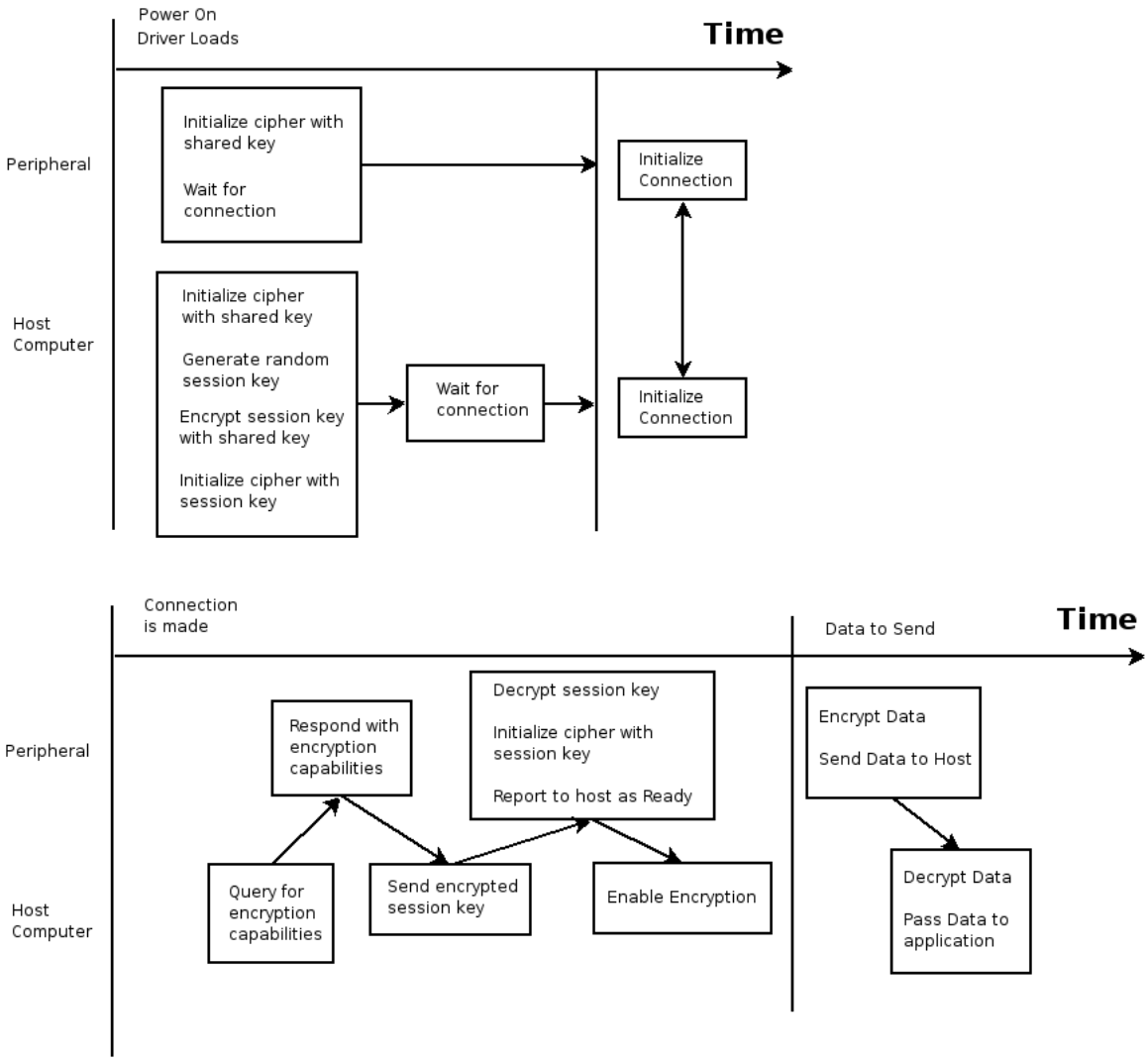

Figure 3-1: Protocol for Initiating and Maintaining Encryption

3.3 Implementations on Specific Interfaces

3.3.1 RS-232

Because of their simple point-to-point nature, RS-232 interfaces typically use very simple protocols for communication. The RS-232 specification does not include any specific protocol detailing specific data bytes to be used (other than Software flow control bytes, which are rarely used); each manufacturer creates their own language for communications. The manufacturer of an RS-232 device that would support this type of encryption would need to assign certain commands, which are not already used in the protocol for other functions, that could be used for control commands, e.g., to query for the encryption capabilities, the response to this query, and the sending of the encrypted session key. The data is encrypted immediately before the peripheral sends it to the transmit buffer of the UART (the UART is the chip that actually sends and receives data over an RS-232 connection). The data is decrypted immediately when it is received by the host computer's UART, before the driver begins to process the data.

3.3.2 USB Implementation

In a USB interface, the existing driver for the device would be modified so that on initialization of the driver, a control pipe is created for setting up the encryption. The query for the encryption capabilities and its response would take place over this control pipe, as would the sending of the encrypted session key. Once the cipher is initialized with the session key in both the host and the peripheral device, all data through the data pipe can then be encrypted normally.

3.3.3 TCP/IP Implementation

As mentioned in Chapter 2, TCP/IP is not commonly used as an interface between computers and their peripherals. Also, there are already protocols available for securing TCP/IP communications. Because of this, an implementation of this work for TCP/IP is not as important as implementations for other interfaces.

Because end-to-end connections are handled by the transport protocols in the TCP stack, the encryption must be done between application layer and the transport layer. Since the network interface's driver is at the bottom of the stack, just above the physical medium, a second driver would be used, instead of modifying the original network interface driver. This driver would appear to the application to be a network interface. When the application implements the API to send data across the network, the second driver is given the data and the destination address. When a TCP connection (or other transport-layer connection) is made, the second driver initializes the cipher, generates the session key, and queries with the remote peripheral for encryption capabilities, and sends the encrypted key to the peripheral. When the second driver receives the control commands used in this process from the peripheral, it does not pass them to the application, but takes the data, interprets it appropriately, and responds to the peripheral as needed. When an actual data message is received, the second driver decrypts the data, and then passes it to the application.

Note that with this design, because the driver is above the transport layer, there would be a separate instance of the encryption system for each simultaneous connection. This makes sense, because one host computer could potentially be communicating with several remote peripherals, some of which could support this encryption system, and

some of which could not, or could support different ciphers. In order for the encryption system to stay synchronized with multiple different devices, a different cipher instance must be maintained for each connection. It is possible that two devices could use more than one connection at a time, and in this case, each of those connections would need a separate cipher.

3.3.4 Bluetooth Implementation

Bluetooth uses multiple different transport protocols, and communicates with multiple different devices, so it would not be practical to have one cipher that handles all protocols and connections simultaneously. As with the TCP/IP implementation, the Bluetooth implementation would need a separate instance for each connection. Also as with the TCP/IP implementation, the encryption would need to be done in a second driver between the application and the transport layer. Like TCP/IP, encryption protocols currently exist in Bluetooth, reducing the need for a new layer of encryption.

3.3.5 Demonstration Implementation

This research has been done by creating a demonstration system to work between a standard PC keyboard and a host computer connected through the PS/2 bus. In the final demonstration system, the keyboard and the microcontroller communicate through the PS/2 bus, but the interface between the microcontroller and the host computer was changed to RS-232, for reasons which will be detailed later. The principal components of this system are the host computer, the keyboard, a microcontroller which encrypts the data from the keyboard, and a software driver loaded in the operating system of the host

PC, which manages the encryption session and decrypts data from the encrypting microcontroller. In a production system, the encryption microcontroller would be built in to the keyboard. It would either be a secondary processor that receives data transmitted by the main processor of the peripheral device, or the encryption functionality and the peripheral's main functionality would all be handled by one processor.

3.3.5.1 Equipment Used

The microcontroller chosen for the demonstration system is a PIC 18F452 from Microchip Technology Inc. The PIC is a relatively simple 8-bit microcontroller with 5 separate I/O ports, an integrated LCD controller, and an integrated RS-232 controller. This microcontroller was chosen for several reasons, including:

1) Large number of I/O pins for debugging
2) Integrated RS-232 controller for debugging (this was later used as the interface for communication between the microcontroller and the host computer)
3) Availability and sophistication of development tools within the department
4) Availability of samples from the manufacturer
5) C Programmability
6) Relatively simple microcontroller allows for more accurate simulation of the simple processors in most peripheral devices

The IDE used to program and debug the microcontroller is Microchip's MPLAB IDE v7.20. The MPLAB C18 Student Edition v2.40 compiler was used to provide C programmability.

Two host computers have been used during the course of this research. One is a Toshiba Satellite notebook (2.66 GHz Pentium 4 processor, 512 MB RAM). The other is a Dell Optiplex GX1 (233 MHz Pentium II processor, 128 MB RAM). The keyboard used is a standard PS/2 keyboard that was bundled with the Optiplex host computer. The operating system used on the host computer is the Ubuntu 5.04 (Hoary Hedgehog release) distribution of the Linux operating system. The gcc C compiler included with the operating system was used to program the software on the host computer that receives the data.

The host computer runs the Ubuntu distribution of the Linux operating system. A microcontroller has been programmed to communicate with a software program (driver) on the host computer. The driver and the microcontroller have a common, hard coded initial key. The driver generates a random session key, encrypts it using the initial key, and then sends encrypted session key to the microcontroller. The driver and the microcontroller each initialize a cipher using this session key, after which all data sent from the keyboard to the host computer is encrypted using the cipher.

3.3.5.2 The PS/2 Protocol

Before the microcontroller can be programmed to intercept data from the keyboard and encrypt it, the PS/2 interface must be properly understood. PS/2 is the interface designed for computer keyboards and mice to communicate with IBM personal

computers (and later, all Intel-based personal computers). PS/2 devices use a 4 wire connection between the peripheral device and the host computer. The wires are:

1) +5V
2) Ground
3) Data
4) Clock

When no data is being sent, the data and clock lines are in a logical high state (+5V). The clock signal is generated by the peripheral device, and is only generated during transmission of each individual data byte. The frequency of the clock signal is 10 – 16.7 kHz. Each data frame consists of a start bit, 8 data bits, an odd parity bit, and a stop bit. When the peripheral device has a byte to send, it pulls the data line low, then it pulls the clock line low, (constituting the start bit) then generates the clock signal at the typical frequency. The data line changes states when the clock is high, and is read by the host computer when the clock is low. After the last bit is sent, the peripheral device completes one more clock cycle while leaving the data line high, which constitutes the stop bit.

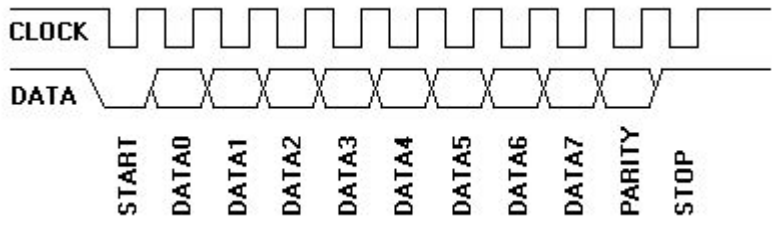

Figure 3-2: Peripheral Transmitting Data (Chapweske, 2003)

When the host computer has a byte to send, it pulls the clock line low and waits at least 100 microseconds, then pulls the data line low. The clock line is then released and is thereafter controlled by the peripheral device, which generates the clock signal. The host changes the state of the data line while the clock is low, and the device reads the state of the data line when the clock is high. After all eight data bits and the stop bit have been transmitted, the data line is released. The peripheral device then pulls the data line low, then the clock line, and then releases both the data and the clock lines. This is an ACK bit, signifying that the peripheral device has received the data, and that no errors were detected.

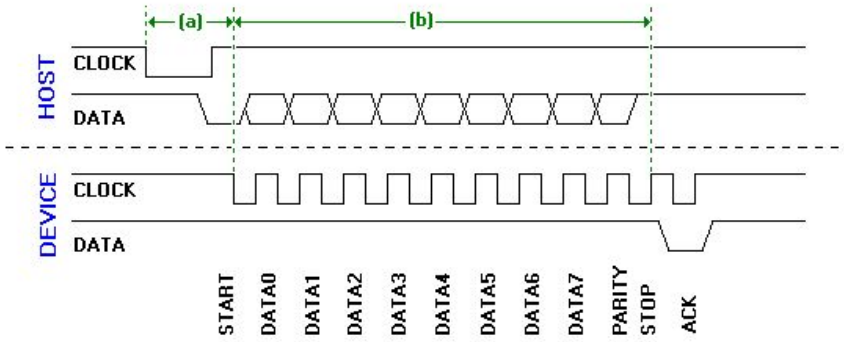

Figure 3-3: Host Transmitting Data (Chapweske, 2003)

The data sent from the keyboard to the host computer is either a scan code or one of a number of predefined commands or status messages. Scan codes are broken down into two types: make codes and break codes. A make code is sent each time a key is pressed, and resent with a frequency determined by the host computer, until the key is released. Once the key is released, the keyboard sends a break code. There are 3 different sets of Scan Codes (2 of which are commonly used). Each set contains codes to

represent each of the keys on a keyboard. Standard keyboards only use codes from Set 2. PC operating systems (Windows, Linux, *BSD, and others) use codes from Set 1. When the keyboard sends a code from Set 2, the PS/2 controller (the hardware in the motherboard which allows the PC to interface with the PS/2 bus) converts the code to the correct code in Set 1, and passes it on to the operating system. The PS/2 controller will only pass on codes received from the keyboard which are within the range of Set 2 codes, or which are within a specific set of control/status codes.

This represents a problem for this encryption system. If the host computer's PS/2 controller receives an encrypted byte which happens to match a scan code from Set 2, it will translate it to a Set 1 code, which would have to be translated back to Set 2 by the decryption software before the code could be decrypted. If the PS/2 controller receives an encrypted byte that does not match any of the scan codes from Set 2, or if the encrypted data it receives corresponds with a command/status messages, it will not pass that data to the operating system's keyboard driver. That data byte will be lost, and the stream ciphers will no longer be synchronized, which means that the driver will no longer be able to correctly decrypt data that is passed from the PS/2 controller. The encryption system described by Daniel Treat (Treat, 2002) is not affected by this situation, because it decrypted the data before the data reached the PS/2 controller.

This could be remedied by the fact that commands are defined in the PS/2 protocol that can instruct the keyboard to transmit codes from Scan Code Set 1 instead of Set 2, and the host computer's operating system can send a command to the controller to disable the translating of codes between sets, so that the controller will pass all data exactly as it is received onto the operating system. However, not all (and possibly not

very few) PS/2 peripherals implement these commands, since the ability to change scan code sets is a feature that is not commonly used. Testing showed that the host computer used in the demonstration does not respond to these commands.

One alternate solution to this problem which would allow the PS/2 bus to be used between the microcontroller and the host computer, would be to send each encrypted byte as a combination of two scan codes which represent the keys for the hexadecimal value of the encrypted byte. For example, if the encrypted byte has a hexadecimal value of 0xF5, the microcontroller would first send the scan code for the 'F' key, followed by the scan code for the '5' key. In this scenario, the keyboard would be sending only the make codes which represent key presses, and no break codes, which represent key releases. This would most likely not be a problem for the PS/2 controller, but this is currently unknown, as it has not been verified through testing.

In order to work around this obstacle, the microcontroller takes the data received from the keyboard on the PS/2 bus and transmits it to the host computer through the microcontroller's built-in RS-232 port. RS-232 is an ideal interface to demonstrate the encryption system, because it has no protocol to define which data bytes can and cannot be transmitted. Also, the hardware supporting RS-232 signaling is integrated into the microcontroller, so the microcontroller doesn't need to control the transmission of individual bits in software. Also, RS-232 ports are very common on desktop computers. At this point, the software running on the computer was changed from a PS/2 keyboard driver to a command line application run by the user. (This change was made for the purposes of the demonstration. In a production system, the decryption would still be handled by a driver in the operating system.) This could at some point be turned into a

keyboard driver. This would allow the operating system to actually use the encrypted RS-232 interface just like a standard keyboard.

3.3.5.3 Choice of Cipher

The RC4 cipher was chosen primarily because of its speed and small memory requirements. RC4 requires only 256 bytes to store the expanded key, and 256 bytes to store the permutated vector of 8-bit numbers. Encryption of each byte takes only enough time for one permutation of the vector (5 simple addition and assignment operations in C), adding two selected numbers from the vector together, and XORing the result with the byte of plaintext. The other ciphers considered, which are all block ciphers, require more processor operations, and have greater memory requirements for each unit encrypted.

3.3.5.4 Demonstration Implementation Process

This program, on initialization, initializes the RC4 cipher, then generates a session key randomly by polling a system timer, and running the result through a hash function. The session key is then encrypted.

When the keyboard reports that it has been powered up (through the BAT code), the driver sends a code to request an encrypted session. If the microcontroller does not respond with the code to accept the encryption session, the driver will just continue to receive unencrypted data from the keyboard, printing it to the screen without trying to decrypt it. If, however, the microcontroller responds with the code to accept the encrypted session, the driver will send the key to the microcontroller one byte at a time.

Once all of the bytes have been received, the microcontroller decrypts the key using the initial key, initializes the cipher with the new session key, then responds to the driver with a code to acknowledge receipt of all of the bytes. At that point, both the driver and the microcontroller begin using the cipher to encrypt and decrypt each byte of data sent from the keyboard to the driver.

The shared key is a potential weakness because if an attacker discovers the key, he or she can decrypt the session key as it is transmitted from the host to the peripheral. This would give the attacker the ability to decrypt all data sent from the peripheral. This key will be relatively safe, however, because it is used very little in each session. Without a large amount of ciphertext to analyze, it would be very difficult for an attacker to discover the session key. A public key exchange protocol such as Diffie-Hellman would eliminate the need for a shared, hard-coded key, but it may not be possible to practically implement such an exchange on a small microcontroller with limited processing power and memory.

Chapter 4

Results and Analysis

4.1 Results

Performance of the system was quantified by measuring the amount of time between the end of reception of the scan code from the keyboard and the beginning of the transmission of the code over RS-232 to the host computer, with encryption turned off, then comparing the results to the same measurement with encryption turned on. The measured amount of time varied with encryption turned on and turned off, but stayed within a reasonably consistent range. In each of the following oscilloscope printouts, the lower waveform represents the communication from the keyboard to the microcontroller (PS/2), while the upper waveform represents communication from the microcontroller to the host computer (RS-232).

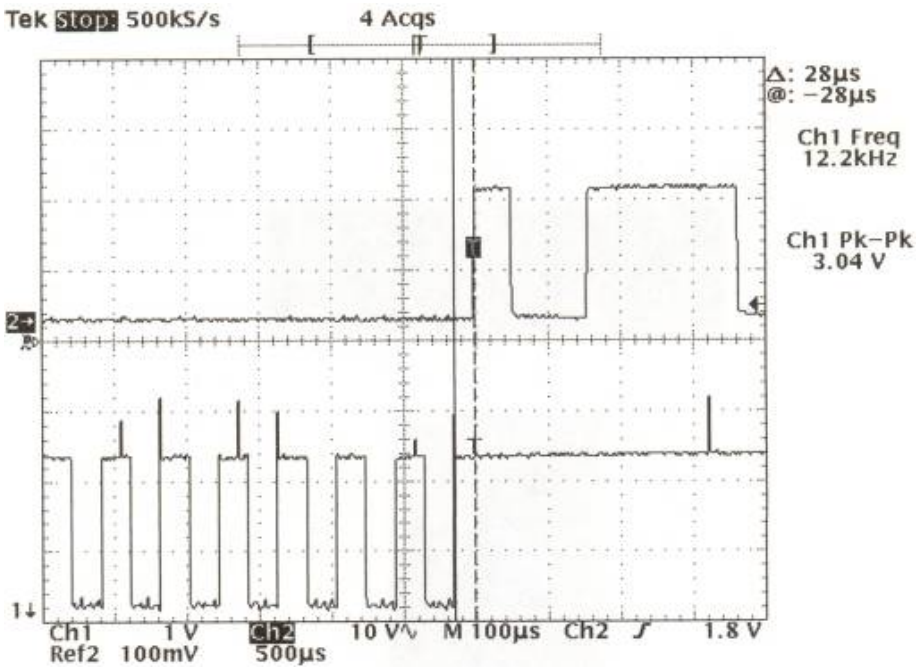

Figure 4-1: Minimum Turnaround Time (Without Encryption)

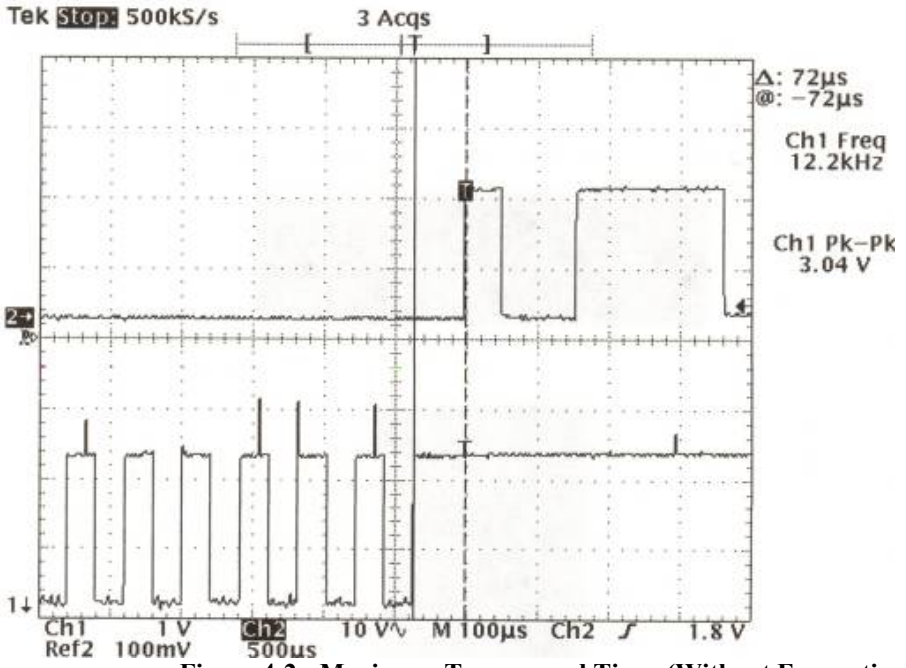

Figure 4-2: Maximum Turnaround Time (Without Encryption)

Figures 4.1 and 4.2 show the minimum and maximum measured times between reception and retransmission of the unencrypted scan code. The end of the reception of the code from the keyboard is marked by the solid vertical line just to the right of the center of the figure. The beginning of the transmission to the PC is marked by the dashed vertical line to the right of the solid line. The time between these two lines is the time it takes for the microcontroller to process the received byte and prepare to begin transmitting it through the RS-232 interface. The label at the top-right corner of each figure with the delta symbol is the measurement, in μs, between the two lines, as measured by the oscilloscope. The difference ranges from 28 μs to 72 μs. Twenty measurements were taken, to establish the range of values. The actual latency added by the microcontroller is this turnaround time added to the time it takes to transmit the scan code through the RS-232 interface. (The time to receive the code from the keyboard through the PS/2 interface is a part of any keyboard PS/2 communication, even without the microcontroller, so it is not included in the added latency.) At 19200 bits per second, the transmission time for one byte is

$$\frac{(1start bit + 8data bits + 1stop bit)}{19200 bits per second} = 521\mu s \qquad \textbf{(4-1)}$$

The total time required to process a received byte, and retransmit it via the RS-232 interface is between 549 μs and 593 μs.

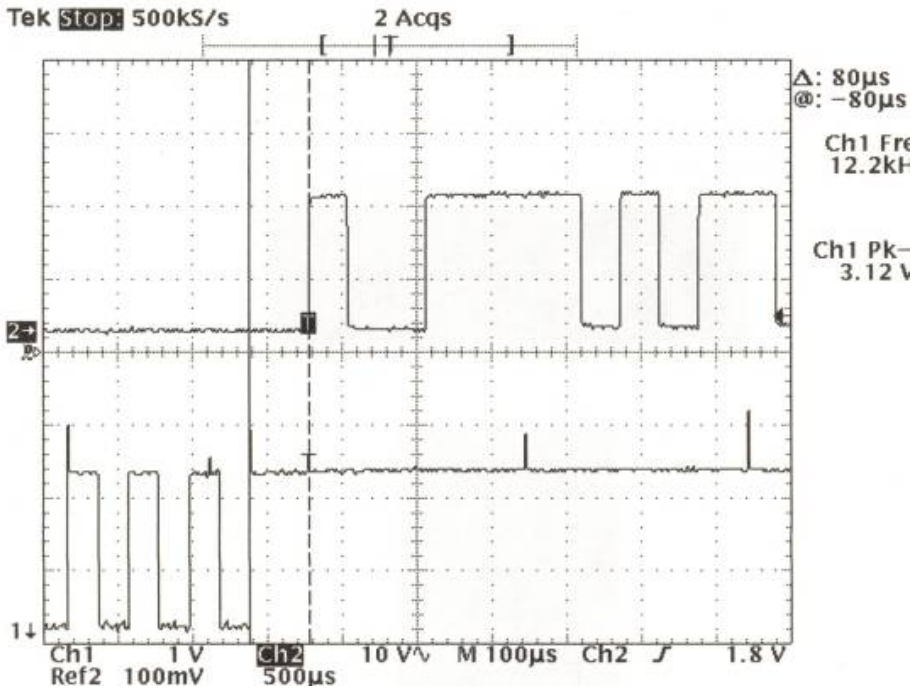

Figure 4-3: Minimum Turnaround Time (With Encryption)

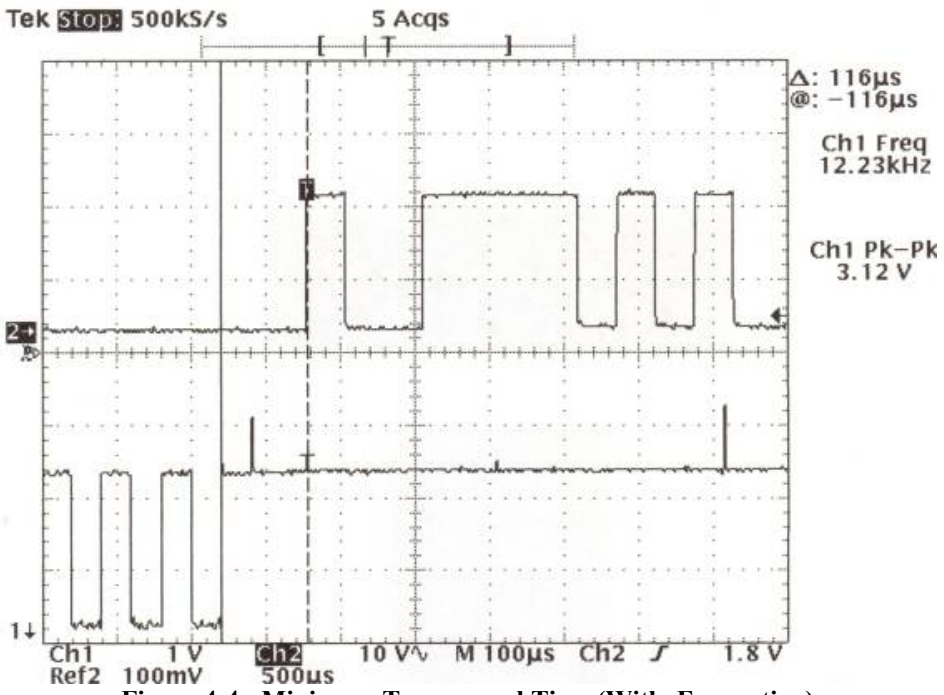

Figure 4-4: Minimum Turnaround Time (With Encryption)

Figures 4.3 and 4.3 show the measured time between reception and retransmission of the encrypted scan codes. (In these figures, the solid and dashed lines appear to the left of the center of the graph.) The minimum measured time is 80 µs, and the maximum time is 116 µs. The total latency is that value added to 521 µs (601 µs to 637 µs).

Table 4-1 Time for Microcontroller to Process and Transmit Data

	Minimum (µs)	**Maximum (µs)**	**Range (µs)**
Without Encryption	549	593	44
With Encryption	601	637	36
Difference (µs)	52	44	

The difference between the minimum times is 52 µs, while the difference between the maximum times is 44 µs. The assembly listing for the encrypt() function is 101 instructions long. (This includes making the function call and returning, in addition to making the call to a function which performs a permutation of the RC4 vector each time the encrypt() function is called.) This differs from the 8 – 16 instruction cycle quote given for RC4 in chapter 4, because the architecture of the PIC microcontroller requires a fair amount of overhead to access the 256-byte vector, which is stored in a separate page of memory.) Each instruction cycle is 4 clock cycles on the PIC. The clock frequency is 11.0592 MHz. This gives an execution time of

$$\frac{4*101 cycles}{} * \frac{1s}{11{,}059{,}200 cycles} = 36.5 \mu s \qquad \textbf{(4-2)}$$

The rest of the 44 – 52 µs can be attributed to overhead from finishing receiving the scan code from the keyboard, preparing to call encrypt(), and passing the resulting encrypted value to the RS-232 transmission function.

The rate at which a scan code is repeated (the typematic rate) is, by default 10.9 repeats per second, but can be as high as 30 repeats per second. The time taken for a keyboard to transmit one code can be anywhere from 720 µs to 1200 µs, depending on the clock speed of the individual keyboard. That time combined with the maximum 52 µs spent encrypting the code and preparing to send it to the host computer, gives a maximum total time of 1252 µs that the microcontroller is busy receiving a code and encrypting it. This time would be less if an integrated PS/2 controller were used to receive the byte, so that the microcontroller would not have to handle reading each individual bit as it is received on the PS/2 interface (instead the microcontroller would read the byte from an internal register after the integrated PS/2 controller received it) or if the keyboard's built-in microcontroller were doing the encryption, so no PS/2 interface would be necessary at all). If the microcontroller is busy for 1252 µs out of every 90 ms, then it is idle for 98.6% of the time. Even considering a worst-case scenario where the repeat rate is set to 30 repeats/second, which means that there would be 33 ms between repeats, (a fast typist types between 5 and 10 keys per second) the microcontroller is still idle 96.2% of the total time.

The total latency for the system would be at most 1200 µs to receive the code from the keyboard, plus 637 µs (the maximum length of time to encrypt the scan code, prepare it for transmitting over RS-232 and the RS-232 transmission time, according to

Table 4.1 above), which totals 1837 µs. A user would not be likely to notice this amount of latency.

4.2 Observed Response Test

A sample group of ten people was asked to use the demonstration system one at a time, without knowing whether the encryption was enabled or disabled. The program shows the user that the encryption system is either in "State G" or "State H," and allows the user to toggle the state using the F1 key, but does not clarify which in state the encryption is turned on, and in which state the encryption is turned off. Each person in the group was asked to type a simple sentence as many times as necessary, toggling between the two states, until he felt that he knew which state responded slower to each key press.

Table 4-2: Observed Response Test Results

Thought "State G" was more responsive	Thought "State H" was more responsive
6	4

State G is the Non-Encrypting state, and State H is the Encrypting State. It would be expected that State G would be more responsive, if there is a noticeable difference in response time, but the results were almost evenly split between G and H. One member of the test group who felt that State G was more responsive said that he could tell no difference between the two, and only chose G because he was required to choose one of the two. The sample size was small, but the fact that nearly as many test subjects chose State H as those who chose State G, indicates that there is no noticeable response time

difference. This is not unexpected given the 44 μs – 52 μs encryption times measured in Figures 4-1 through 4-4. 52 μs is too short for a person to be able to notice any difference.

Chapter 5

Conclusions and Recommendations

The proposed system provides a simple and fast way to encrypt data between a host computer and its peripheral devices, allowing for secure transmission of potentially sensitive data. This system involves a microcontroller within the peripheral device which encrypts the data, and a driver on the host computer which decrypts the data. The system is simple enough that microcontrollers with very small memory capacities and slow clock speeds can still use it. The proposed system is easily adaptable to work over a variety of different interfaces. Both wired and wireless interfaces can be supported.

The RC4 stream cipher provides adequate encryption for peripherals, both in terms of being a strong cipher, and in terms of efficient processor utilization. Applications such as a keyboard, which deal with one byte of data at a time, are very good candidates for RC4 because RC4 is oriented towards encrypting one individual byte at a time. Other peripherals, such as biometric scanners, which would send a complete set of data at once, would be more efficiently served with a block cipher, which would encrypt larger groups of data at one time.

One suggestion for future research would be to explore using block ciphers for peripherals which transmit data in chunks. AES would most likely be the cipher of choice, but others could also be explored. Another avenue for future research would

involve using public-key encryption for the initial key exchange. Using public-key encryption, the host computer and the peripheral device would each be able to verify the identity of the other before beginning encryption, and they would not have to rely on a "secret" shared key, which, if it were discovered, would make all of the encryption worthless. Public-key encryption would be a challenge, because one would need to implement libraries that would allow for multiple-precision arithmetic on the embedded microcontroller, and because the larger keys and the multiple precision math require more RAM. The GNU Multiple Precision library (GNU MP) would be suitable for use on the host computer, but finding a suitable library for the microcontroller may prove to be more difficult. Doing public-key encryption on such a small processor would take a noticeably long amount of time, but once the initial key is exchanged and the symmetric cipher is initialized, the user would not notice any additional delays.

Another question for additional research is whether periodically generating new keys would add or detract from the security of the system. If the system is routinely exchanging newly generated keys, would that expose a point of weakness in the key exchange process or in the initial key that an attacker could exploit, or would it make it more difficult to break the encryption because each set of data encrypted with the same session key is smaller?

A final suggestion for further investigation is to study the relationship between encryption performance and microcontroller processing power in greater depth. While this work used one cipher which is known for being simple and fast, further study should measure and compare the amount processing power required by different ciphers running

on 8-bit microcontrollers. The effects of integrating the encryption with the workloads of the existing microcontrollers in peripherals should be taken into account.

The system proposed in this work is just one step toward total system security, and is not intended as a guarantee that a computer system is completely secure. Measures such as physical access restriction, strong password policies, and other steps recommended by the security community will always be vital for secure applications.

References

Allen Concepts, 2003 http://www.keykatcher.com/legal/index.htm, Retrieved 4/5/2005

Atasu, K., Breveglieri, L., Macchetti, M., (2004) Efficient AES implementations for ARM based platforms., Proceedings of the ACM Symposium on Applied Computing, v 1, pp 841 – 845.

Bluetooth SIG., (2006) The Official Bluetooth Wireless Info Site., http://www.bluetooth.com/bluetooth/ Retrieved 12/20/2005.

Bolles, G., (2002) Wireless Network Security, Technology: Wireless, CIO Insight., http://www.cioinsight.comprint_article0,3668,a=29387,00.asp. Retrieved 2/13/2006

Brandt, A., 2003, http://www.pcworld.com/howto/article/0,aid,108712,00.asp Keyboards that Blab, Retrieved 11/25/2005

Chapweske, A., 2003 http://www.Computer-Engineering.org, Retrieved 3/15/2003

Crowcroft, J., Phillips, I., TCP/IP and Linux Protocol Implementation: Systems Code for the Linux Internet, John Wile & Sons, Inc., New York, Publisher: Robert Ipsen, 2002, pp 62 – 76.

Felten, E.W., (2003) Understanding trusted computing: will its benefits outweigh its drawbacks?., IEEE Security & Privacy Magazine, IEEE, v 1 n 3, pp 60 – 62.

Fluhrer, S. R., Mantin, I., Shamir, A., (2001) Weaknesses in the Key Scheduling Algorithm of RC4. Selected Areas in Cryptography 2001, pp1–24.

Frerking, G., Baumann, P., (2001) Serial Programming HOWTO., http://www.faqs.org/docs/Linux-HOWTO/Serial-Programming-HOWTO.html, (Retrieved 3/15/2005)

Goodman, J., Dancy, A.P., Chandrakasan, A.P, (1998) An Energy/Security Scalable Encryption Processor Using an Embedded Variable Voltage DC/DC Converter., IEEE Journal of Solid-State Circuits, v 33, n 11, pp 1799 – 1809.

Hager, C. T., Midkiff, S. F., (2003) Demonstrating Vulnerabilities in Bluetooth Security., Conference Record / IEEE Global Telecommunications Conference., v 3., pp 1420 – 1424.

Huang, A., (2005) An Introduction to Bluetooth programming in GNU/Linux., http://people.csail.mit.edu/albert/bluez-intro/ (Retrieved 12/21/2006)

Huang, A., (2005) The Use of Bluetooth in Linux and Location Aware Computing., Master's Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology.

Kelsey, J., Schneier, B., Wagner, D., (1997) "Related-Key Cryptanalysis of 3-WAY. Biham-DES, CAST, DES-X, NewDES, RC2, and TEA", International Conference on Information and Communications Security, Beijing, Springer-Verlag, pp. 233-246.

Kundarewich, P.D., Wilton, S. J.E., Hu, A. J., (1999) A CPLD- Based RC-4 Cracking System., The 1999 Canadian Conference on Electrical and Computer Engineering., May 1999

Lawrence, E., Lawrence, J., (2004) Threats to the Mobile Enterprise: Jurisprudence Analysis of Wardriving and Warchalking., International Conference on Information Technology: Coding Computing, ITCC 2004, pp 268-273.

Luo, X., Zheng, K., Pan, Y., Wu, Z., (2004) Encryption algorithms comparisons for wireless networked sensors., Conference Proceedings - IEEE International Conference on Systems, Man and Cybernetics, pp 1142 – 1146.

Monkman, R., (2002) Enhancing Embedded Security., EDN, v 47, n 23, pp 61 – 66.

Mueller, R. S., 1992, http://doe-is.llnl.gov/Orders/dojkeymn.pdf, Retrieved 4/5/2003

Potter, B., (2005) Bluetooth Attackes Start to Bite., Network Security., V 2005, n 2, pp 14 – 15.

Raytown Corp, 2005 http://www.keyloggers.com/bigbrother.html, Retrieved 4/4/2005

Rivest, R., (2004) RSA Security Response to Weaknesses in Key Scheduling Algorithm of RC4., http://www.rsasecurity.com/rsalabs/node.asp?id=2009 , (Retrieved 1/23/2004).

Schneier, B., (2005) Schneier On Security: Trusted Computing Best Practices., http://www.schneier.com/blog/archives/2005/08/trusted_computi.html (Retrieved 12/15/2005).

Schneier, B., (1995) The Blowfish Encryption Algorithm – One Year Later., Dr. Dobb's Journal, 234, pp 137 – 138.

Schneier, B., (1993) Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish)., Fast Software Encryption, Cambridge Security Workshop Proceedings, (December 1993), Springer-Verlag, pp. 191-204.

Shaked, Y., Wool, A., (2005) Cracking the Bluetooth PIN., Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services Mobisys '05, pp 39 – 50.

Stallings, W., (2002) Cryptography and Network Security: Principles and Practice (3rd Edition)., Prentice Hall.

Strangio, C. E., 1993-2006 by CAMI Research Inc., http://www.camiresearch.com/Data_Com_Basics/RS232_standard.html, Retrieved 2/13/2006

Su, C., Lin, T., Huang, C., (2003) A High-Throughput Low-Cost AES Processor., IEEE Communications Magazine, v 41, n 12, pp 86-91.

Treat, D.G.; Keyboard encryption Potentials, IEEE, Volume 21, Issue 3, Aug-Sep 2002 Page(s):40 – 42, Digital Object Identifier 10.1109/MP.2002.1033666

USB Implementers Forum, Inc., Universal Serial Bus Specification, http://www.usb.org/developers/docs/ , 2000, pp 34 – 36, Retrieved 2/13/2006

Webb, W., (2004) Hack This., EDN, v 49, n 15, pp. 26 – 8, 30, 32, 34.

Wong, S., (2003) The Evolution of Wireless Security in 802.11 Networks: WEP, WPA and 802.11 Standards. GSEC Practical v1.4b.

Appendices

Appendix A

Source Code: PIC Microcontroller

A.1 pic_enc.h

```
#define kbdpt PORTBbits.RB2
#define kbdclk PORTBbits.RB1

#define start_enc 0xDD
#define stop_enc 0xDC
#define accept_enc 0x01
#define keyReceived 0xDA
#define NEXT_KEY_CHAR 0x99
#define PC_ACK 0xFA
#define RESEND 0xFE
#define BAT 0xAA

#define RESET 0xFF
#define setTypematicDelay 0xF3
#define readDeviceID 0xF2
#define setScanCodeSet 0xF0
#define echo 0xEE
#define setLEDs 0xED

#define baud19200 35

// matched key:  82AB9D0C5872291B

#define Ka0 0x82
#define Ka1 0xAB
#define Ka2 0x9D
#define Ka3 0x0C
#define Ka4 0x58
#define Ka5 0x72
#define Ka6 0x29
#define Ka7 0x1B

#define wait1800us Delay1KTCYx(5) // 5000 cycles
#define wait100us Delay10TCYx(28) // 276 cycles ~ 280 cycles
#define wait40us Delay10TCYx(11) // 111 cycles  ~ 110 cycles
#define wait20us Delay10TCYx(6) // 55 cycles ~ 60 cycles
#define wait10us Delay10TCYx(3) //27 cycles ~ 30 cycles

#define keylength 8

#define dis_int INTCONbits.GIE = 0;
```

```
#define en_int INTCONbits.GIE = 1;

#define hinder_kbd TRISB &= 0xFD; kbdclk = 0    //sets kbdclk to 0,
sets it as an output
#define allow_kbd kbdclk = 1; TRISB |= 0x02     //sets kbdclk to 1,
sets it as an input
//#define cycle istream++; jstream += S[istream]; tmp = S[istream];
S[istream] = S[jstream]; S[jstream] = S[tmp]

far unsigned char istream, jstream = 0;
far unsigned char encryption_on = 0;

unsigned char K[keylength];
unsigned char SesKey[8];

unsigned char scan_code_set = 2;

int keyRXMode = 0;
int next_byte = 0;

void setupInts(void);

unsigned char enc_signal = 0;

unsigned char kbd_buf[8] = {0x99, 0x99, 0x99, 0x99, 0x99, 0x99, 0x99,
0x99};
unsigned char bufcount = 0;
unsigned char trash;
unsigned char BATcnt = 0;
unsigned char init_level;

#pragma udata S_array
unsigned char S[256];

#pragma udata T_array
unsigned char T[256];

void rxpcpt(void);
unsigned char rxkbdpt(void);
int txkbdpt(unsigned char txbyte, int count);
void txpcpt(unsigned char txbyte);
unsigned char readpcpt();
void getkbdptbyte(void);
void hinder(void);
void buffer(unsigned char to_send);
void cycle(void);


void init_T ( void ); // S[i] = i; T[i] = K[i % keylength]
void init_S ( void ); // j=(j+S[i]+T[i]); tmp=S[i]; S[i]=S[j]; S[j]=tmp

// Set up initial permutation of RC4, based on the key, K
void initRC4(void);

// Do each RC4 round's permutation and do the encryption
unsigned char encrypt(unsigned char mcode);
```

```
#pragma code highVector=0x08
void athighVector (void) { _asm GOTO hinder _endasm}
#pragma code
#pragma interrupt hinder
```

A.2 pic_enc.c

```
/*********************************************************
**    Kelly Norman 2003 - 2006
**    This program takes input directly from a
**    keyboard's PS/2 port, and passes it to the
**    PIC microcontroller's RS-232 port.  The program
**    also listens for messages from the host computer
**    requesting an encrypted session, echanges session
**    keys with the host computer, and encrypts data using
**    the RC4 stream cipher for secure data transmission
**    to the host computer.
**
**    The data coming from the keyboard is in the format
**    of PS/2 Scan Codes (Set 2) and PS/2 commands.
**    Some of the commands have been created for this
**    this program, and are not a part of the original
**    PS/2 protocol
**
**    http://www.computer-engineering.org , built and
**    maintained by Adam Chapweske, has been a tremendous
**    source of information on the PS/2 protocol, by far
**    the most useful resource on the Internet, and the
**    only resource for the vast majority of the information
**    I needed.
*********************************************************/


#include<p18f452.h>
#include<delays.h>
#include<usart.h>
#include"pic_enc.h"

void main(void)
{
  unsigned char a;
      int i;
//  unsigned char b = 0;

  OpenUSART(USART_TX_INT_OFF &
      USART_RX_INT_ON &
      USART_BRGH_HIGH &
      USART_EIGHT_BIT &
      USART_CONT_RX &
      USART_ASYNCH_MODE, baud19200);

  // Set up initial key, used to decode the session key, which will be
sent by the keyboard later
  K[0] = Ka0;
  K[1] = Ka1;
  K[2] = Ka2;
  K[3] = Ka3;
  K[4] = Ka4;
  K[5] = Ka5;
  K[6] = Ka6;
  K[7] = Ka7;
```

```
  setupInts();    //Prepare and enable interrupts
  en_int;
  allow_kbd;            //ensure the kbdisable to send if it needs to
  initRC4();

  txpcpt( BAT );

  while (1)
  {
    if(bufcount != 0)
    {       //if any bytes need to be sent to thekbd, send them now
      dis_int;          //disable intstoprotect global kbd_buf and
bufcount
      txkbdpt(kbd_buf[0], 0); //tx the first byte in the buffer
      bufcount--;       //then shift the rest of the bytes down one
      for(a=0; a<7; a++)
      {
          kbd_buf[a] = kbd_buf[a+1];
      }
      en_int;           //re-enable interrupts
    }
    else
    {
      allow_kbd;  //ensure the kbd is able to send if it needs to
      if (!kbdpt) //get byte from kbd if it is sending one
      {
          dis_int;
        getkbdptbyte();
          en_int;
      }
    }
  }
}

void setupInts(void)
{
  INTCONbits.GIE = 1;
  INTCONbits.PEIE = 1;
  IPR1bits.RCIP = 1;
}

void getkbdptbyte(void)
{
  txpcpt(rxkbdpt());
}

unsigned char readpcpt(void)
  // This procedure waits until a byte has been received by the UARt,
  // then gets it, sends a diagnostic message through the UART,
  // and returns the value of byte received
{
  unsigned char rxByte;

  //Wait until a byte is ready.  If called by an interrupt, it will
already be true,
  //but if we're reading the port when int's are disabled, we may have
to wait.
```

```
  while(!DataRdyUSART());
  rxByte = ReadUSART();
  PIR1bits.RCIF = 0;

  return rxByte;
}

void rxpcpt(void)
  //This procedure receives a byte from the PC.  Depending on the byte
received,
  //it may also wait for a second byte.  Then it will enable keyRXMode,
pass on the
  //byte(s) to the keyboard, or respond directly to the PC as
appropriate.  If it
  //is currently in keyRXMode, it will read the key bytes into the key
array, then
  //turn off keyRXMode.
{
  unsigned char rxbyte1;
  unsigned char rxbyte2;
  unsigned char rxbyte3;
      unsigned char sum = 0;
  int i;

  TRISB = 0x8D;         // 1000 1101 Set kbdclk to output, the rest to
inputs
  kbdclk = 0;           // Prevent kbd from sending codes

  //if the PC is not sending encrypted session key bytes, then just get
the regular bytes
  rxbyte1 = readpcpt();
  switch(rxbyte1)
  {               //respond to different commands from the PC
differently
    case NEXT_KEY_CHAR:
        rxbyte2 = readpcpt();
        rxbyte3 = readpcpt();
        K[ rxbyte2 ] = rxbyte3;
      // Now we receive the whole key at once

        if( rxbyte2 == keylength - 1 )
        {
                  for( i = 0; i < keylength; i++ )
                  {
                        K[ i ] = encrypt( K[ i ] );
                        sum += K[ i ];
                  }
          initRC4();
          txpcpt( keyReceived );
          encryption_on = 1;
        }
        break;
      case start_enc:
        txpcpt(accept_enc);
        break;
      case stop_enc:
        encryption_on = 0;
```

```
        // Set up initial key, used to decode the session key, which
will be sent by the keyboard later
        K[0] = Ka0;
        K[1] = Ka1;
        K[2] = Ka2;
        K[3] = Ka3;
        K[4] = Ka4;
        K[5] = Ka5;
        K[6] = Ka6;
        K[7] = Ka7;

        initRC4();
        break;
      case RESET:       //RESET
        buffer(RESET);
        break;
      case setTypematicDelay:       //SET TYPEMATIC DELAY/REPEAT
        buffer(setTypematicDelay);
        buffer(readpcpt());
        break;
      case readDeviceID:            //READ DEVICE ID
        buffer(readDeviceID);
        break;
      case setScanCodeSet:          //SET SCAN CODE SET
        buffer(setScanCodeSet);
        buffer(readpcpt());
        break;
      case echo:        //ECHO
        txpcpt(echo);   //send echo byte back to PC
        break;
      case setLEDs:           //SET LEDs FOR NUM/CAPS/SCROLL LOCK
        buffer(setLEDs);
        rxbyte2 = readpcpt();
        buffer(rxbyte2);
        break;
    default:
        break;
  }

  kbdclk = 0;
  TRISB = 0x8D;   //1000 1101  Prevent kbd from sending
}

int txkbdpt (unsigned char txbyte, int count)
      // txbyte is the code to be passed to the keyboard
      // count is the recursion depth of the function,
{
  int i;
  unsigned char bits[9];
  unsigned char KBD_ACK;

  bits[8] = 1;
  if (count >= 3) return 0;

  TRISB = 0x89;   // 1000 1001  kbdclk and kbdpt are outputs, all
others are inputs
  kbdclk = 0;
```

```
  // Set up bits to send before starting to send
  for(i=0; i<8; i++) {
    bits[i] = (txbyte >> i) & 0x01;
  }

  for(i=0; i<8; i++) {
    bits[8] ^= bits[i];
  }

  wait100us;
  kbdpt = 0;
  wait20us;
  kbdclk = 1;
  TRISB = 0x8B;   // 1000 1011  return control of kdbclk to the
keyboard

  //send 8 data bits + parity
  for(i=0; i<9; i++) {
    while (kbdclk);
    kbdpt = bits[i];
    while (!kbdclk);
  }

  while (kbdclk);
  kbdpt = 1;
  TRISB = 0x8F;  // 1000 1111b
  while (!kbdclk);
  while (kbdclk);

  wait10us;

  //check for ACK
  if (kbdpt)
      //ACK not received ... call txkbdpt again recursively
  {
    wait20us;
  }
  else
  {
    while(!kbdpt);      //kbdclk is already high by the time the ACK is
received,
    wait20us;
    TRISB &= 0xFD;      //Set kbdclk to input
    kbdclk = 0;         //Pull kbdclk low for 30us (because the
computer does it, I don't know why
    wait40us;
    TRISB |= 0x02;
                  //so we can go straight into rxkbdpt()
    KBD_ACK = rxkbdpt();
    if(KBD_ACK == 0xFA)
    {
      return 1;
    }
    else if (KBD_ACK == 0xFE )
    {
    }
```

```
  }
  return 0;
}

unsigned char rxkbdpt(void)
{
  int i;
  unsigned char paritybit = 1;
  unsigned char rxbyte = 0;
  unsigned char bits[9];

  while (kbdclk);

  for(i=0; i<9; i++)
  {         // Receive 8 data bits
    while (!kbdclk);
    while (kbdclk);
    wait10us;
    bits[i] = kbdpt;    // receive bit
    if(i<8) paritybit ^= kbdpt;
  }

  if (paritybit == bits[8])   //receive parity bit and check it -- if
good, send code to pc
  {
    rxbyte = bits[0] | bits[1] << 1 | bits[2] << 2 | bits[3] << 3 |
bits[4] << 4 | bits[5] << 5 | bits[6] << 6 | bits[7] << 7;
      //reconstruct byte from individualbits
    if(rxbyte != PC_ACK)
    {
      return rxbyte;
    }


  } else
  {
    en_int; //Give the PC a chance to transmit here if it wants to
    dis_int;
    txkbdpt(RESEND, 0);
  }
}

void txpcpt(unsigned char txbyte)
{
  if (encryption_on)
  {
    txbyte = encrypt(txbyte);
  }
     while( BusyUSART() );
  WriteUSART(txbyte);
}

void hinder(void)
  //This procedure disables interrupts,
{
  unsigned char hinder_msg[] = "Hindered!";
  unsigned char allow_msg[] = "Allowed!";
```

```
  rxpcpt();
}

void buffer(unsigned char to_send)
{
  kbd_buf[bufcount] = to_send;  //bufcount tells the next available
position in the queue
  bufcount++;     //not the last space being used
}

void initRC4(void)
{
  unsigned char i, j, tmp;
  init_T();
  init_S();

  istream = 0; jstream = 0;

  //  Do over 256 rounds of RC4 as per Ron Rivest's reccomendation --
we don't care about the returned values
  for (i=0; i<255; i++) { cycle(); }
  for (i=0; i<64; i++) { cycle(); }
}

void init_T ( void )
{
  unsigned char i = 0;

  while( 1 )
  {
    S[i] = i;
    T[i] = K[i % keylength];
    if ( i == 255 )
    {
      break;
    }
    i++;
  }
}

void init_S ( void )
{
  unsigned char i = 0;
  unsigned char j = 0;
  unsigned char tmp;

  while( 1 )
  {
    j=(j+S[i]+T[i]);
    tmp=S[i];
    S[i]=S[j];
    S[j]=tmp;
    if ( i == 255 )
    {
      break;
    }
    i++;
```

```
  }
}

unsigned char encrypt(unsigned char mcode)
{
  unsigned char tmp;

  cycle();
      tmp = (S[istream] + S[jstream]);  // %256 isn't needed because
this is a char... it can't be more than 256 anyway
  return mcode ^= S[ tmp ];
}

void cycle(void)  //does the same thing as encrypt(), but without
XORing with a variable and returning the variable
{                 //this allows me to save a few cycles while going
through the rounds of the cipher
  unsigned char tmp;

  istream++;  // %256 isn't needed because this is a char... it can't
be more than 256 anyway
  jstream += S[istream];   // %256 isn't needed because this is a
char... it can't be more than 256 anyway

  tmp = S[istream];
  S[istream] = S[jstream];
  S[jstream] = S[tmp];
}
```

Appendix B

Source Code: Linux Application

B.1 decrypt.h

```
#define COMA "/dev/ttyS16"
#define COMB "/dev/ttyS17"
#define COMC "/dev/ttyUSB0"
#define COMD "/dev/ttyUSB1"

#define baudRate B19200
#define _POSIX_SOURCE 1

#define TRUE 1
#define FALSE 0

#define start_enc 0xDD
#define stop_enc 0xDC
#define accept_enc 0x01
#define key_byte 0xCB
#define keyReceived 0xDA
#define NEXT_KEY_CHAR 0x99
#define PC_ACK 0xFA
#define RESEND 0xFE
#define BAT 0xAA
#define RESET 0xFF
#define setTypematicDelay 0xF3
#define readDeviceID 0xF2
#define setScanCodeSet 0xF0
#define echo 0xEE
#define setLEDs 0xED
#define ScrollLockLED 0x01
#define NumLockLED 0x02
#define CapsLockLED 0x04

//matched key: 82AB9D0C5872291B
#define Ka0 0x82
#define Ka1 0xAB
#define Ka2 0x9D
#define Ka3 0x0C
#define Ka4 0x58
#define Ka5 0x72
#define Ka6 0x29
#define Ka7 0x1B
```

```
#define keylength 8

unsigned char istream = 0;
unsigned char jstream = 0;
unsigned char encryption_on = 0;

unsigned char K[keylength];
unsigned char SesKey[8];

unsigned char S[256];
unsigned char T[256];

unsigned char LEDindicators = 0;

unsigned char scanCode[ 144 ] =
{
  0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0A,
0x0B, 0x0C, '\t', '~', 0x0F,
  0x10, 0x11, 0x12, 0x13, 0x14, 'q', '1', 0x17, 0x18, 0x19, 'z', 's',
'a', 'w', '2', 0x1F,
  0x20, 'c', 'x', 'd', 'e', '4', '3', 0x27, 0x28, ' ', 'v', 'f', 't',
'r', '5', 0x2F,
  0x30, 'n', 'b', 'h', 'g', 'y', '6', 0x37, 0x38, 0x39, 'm', 'j', 'u',
'7', '8', 0x3F,
  0x40, ',', 'k', 'i', 'o', '0', '9', 0x47, 0x48, '.', '/', 'l', ';',
'p', '-', 0x4F,
  0x50, 0x51, 'Q', 0x53, '[', '=', 0x56, 0x57, 'C', 0x59, '\n', ']',
0x5C, '\\', 0x5E, 0x5F,
  0x60, 0x61, 0x62, 0x63, 0x64, 0x65, '\b', 0x67, 0x68, 0x69, 0x6A,
0x6B, 0x6C, 0x6D, 0x6E, 0x6F,
  0x70, 0x71, 0x72, 0x73, 0x74, 0x75, 0x76, 'N', 0x78, 0x79, 0x7A,
0x7B, 0x7C, 0x7D, 0x7E, 0x7F
};

#define LShift 0x12
#define RShift 0x59
#define Quote 0x52
#define CapsLock 0x58
#define NumLock 0x77
#define ScrollLock 0x7E
#define LCtrl 0x14
#define LAlt 0x11
#define Num7 0x6C
#define Num4 0x6B
#define Num1 0x69
#define Num8 0x75
#define Num5 0x73
#define Num2 0x72
#define Num0 0x70
#define Num9 0x7D
#define Num6 0x74
#define Num3 0x7A
#define NumDot 0x71
#define NumMinus 0x7B
#define NumPlus 0x79
#define Esc 0x76
```

```
#define F1 0x05
#define F2 0x06
#define F3 0x04
#define F4 0x0C
#define F5 0x03
#define F6 0x0B
#define F7 0x83
#define F8 0x0A
#define F9 0x01
#define F10 0x09
#define F11 0x78
#define F12 0x07
#define ScrLck 0x7E

void initRC4 ( void );
unsigned char decrypt( unsigned char mcode );
void cycle ( void );
unsigned char getKey( unsigned char sCode );

volatile int STOP=FALSE;
void signal_handler_IO (int status);
int wait_flag = TRUE;
struct termios options;
int isLetter( unsigned char asciiByte );
int isNumber( unsigned char asciiByte );
int getComPort();
void InitiateEncryption( unsigned char* key );
int readPort( unsigned char *Buffer);
```

B.2 decrypt.c

```
/*****************************************************
**    Kelly Norman 2003 - 2006
**    This program takes input from a keyboard on the
**    Serial (RS-232) port of a PC running Linux, and
**    initiates and maintains an encrypted channel of
**    communication with the keyboard, using the RC4
**    stream cipher.
**
**    The data coming from the keyboard is in the format
**    of PS/2 Scan Codes (Set 2) and PS/2 commands.
**    Some of the commands have been created for this
**    this program, and are not a part of the original
**    PS/2 protocol
**
**    http://www.computer-engineering.org , built and
**    maintained by Adam Chapweske, has been a tremendous
**    source of information on the PS/2 protocol, by far
**    the most useful resource on the Internet, and the
**    only resource for the vast majority of the information
**    I needed.
**
**    This program also uses code directly from the Linux
**    Serial Programming HOWTO, by Gary Frerking and
**    Peter Baumann,
**    http://www.faqs.org/docs/Linux-HOWTO/Serial-Programming-
HOWTO.html **     specifically nearly the entire initRS232() function,
**    and the sections dealing with interrupts
*****************************************************/

#include <stdio.h>   /* Standard input/output definitions */
#include <stdlib.h>
#include <string.h>  /* String function definitions */
#include <unistd.h>  /* UNIX standard function definitions */
#include <fcntl.h>   /* File control definitions */
#include <errno.h>   /* Error number definitions */
#include <termios.h> /* POSIX terminal control definitions */
#include <sys/signal.h>
#include <sys/types.h>
#include <time.h>
#include <sys/time.h>
#include "decrypt.h"         /* Program-specificinfo */

int fd;
unsigned char newKey[ 8 ] = { 0, 0, 0, 0, 0, 0, 0, 0 };
unsigned char keyCount = 0;
unsigned char sum = 0;
int shift_held = 0;

int main ( void )
{
  int c, res;
  int index = 0;
  unsigned char scanCodeBuffer[32];
  unsigned char tempBuffer[128];
```

```
  unsigned char lastKey = 0;
  int i;
  int shift;
  unsigned char dataToSend[ 6 ] = { 0, 0, 0, 0, 0 };
  unsigned char thisCode = 0;
  unsigned char R1;
  unsigned char R2;
  unsigned char R3;
  int totalBufLen = 0;

  printf("Program Started\n");
  fd = initRS232();

  printf("RS-232 initialized!\n");

  K[ 0 ] = Ka0;
  K[ 1 ] = Ka1;
  K[ 2 ] = Ka2;
  K[ 3 ] = Ka3;
  K[ 4 ] = Ka4;
  K[ 5 ] = Ka5;
  K[ 6 ] = Ka6;
  K[ 7 ] = Ka7;

  initRC4();

  getNewKey();

  printf("RC4 Initialized!\n");

  LEDindicators = 0x02;   //Set LED indicators to NumLock On only

  printf( "Session Key Initialized\n" );

  dataToSend[ 0 ] = RESET;
  keyCount = 0;

  if ( write( fd, dataToSend, 1 ) != 1)
  {
    printf( "sending Reset command failed!\n" );
  }
  else
  {
    printf( "sending Reset command succeeded!\n" );
  }
  /* loop forever, or until the F10 scan code is received on the RS-232
port */
  while (STOP == FALSE )
  //As long as we're not exiting the program, wait for input, and
process data as it arrives
  {
    if ( wait_flag == FALSE )
    //As long as there is data in the receive buffer, process it, don't
wait for more
    {
      res = read( fd, tempBuffer, 255 );
```

```
//read the Serial port buffer until is epmty
if( res > 0 )
//If any bytes are actually read, put them in the buffer
{
  for( i = 0; i < res; i++ )
  {
    scanCodeBuffer[ i + totalBufLen ] = tempBuffer[ i ];
  }
  totalBufLen += res;
  scanCodeBuffer[ totalBufLen ] = 0;
}

// received bytes are all decrypted immediately after reception
// if the encryption_on flag is set
if ( encryption_on )
{
i = totalBufLen - res;

while ( i < totalBufLen )
{
  scanCodeBuffer[ i ] = decrypt( scanCodeBuffer[ i ] );
  i++;
}
}

while( totalBufLen > 0 && STOP == FALSE && wait_flag == FALSE )
{
switch ( scanCodeBuffer[ index ] )
{
        case BAT:
    if( encryption_on == 0 )
    {
      printf( "Keyboard Online!\n\n" );
      LEDindicators = NumLockLED;
      dataToSend[ 0 ] = setLEDs;
      dataToSend[ 1 ] = LEDindicators;
      dataToSend[ 2 ] = 0;
      if ( write( fd, dataToSend, 2 ) != 2)
      {
      printf( "sending LED command failed!\n" );
      }
      else
      {
      printf( "LED command sent succesfully!\n" );
      }
      InitiateEncryption( newKey );
      shift = 1;
    }
    break;

  case accept_enc:
  // After receiving accept_enc
    //      Send new Key to Keyboard (encrypted with old key)
  //  Wait for response from
  //  Turn on encryption_on

    if( encryption_on == 0 )
```

```
          {
            int j;
            for( j = 0; j < keylength; j++ )
            {
            SendKeyChar( j );
            }
            shift = 1;
          }

          break;

        case keyReceived:
          //The keyboard has sent a code indicating that it has
received
          //the entire session key
          printf( "\n\n\t State: H\n\n" );
          encryption_on = 1;
          shift = 1;
          break;

        case 0xF0:
          // Normal BreakCode
          if ( scanCodeBuffer[ index + 1 ] == 0 )     //Complete
scancode has not been received
          {
              shift = 0;
            wait_flag = TRUE;
          }
          else
          {
            thisCode = getKey( scanCodeBuffer[ index + 1 ] );
            if( thisCode == LShift || thisCode == RShift )
            {
            shift_held = 0;
            }
            if ( isLetter( thisCode) )
            {
              if ( LEDindicators & CapsLockLED )
                   //if Caps Lock is on, convert key to Upper Case.
                   //Eventually fix it so that if Shift key is pressed
                   //the same thing will happen
            {
                 thisCode -= 0x20;
            }
            }
            //printf( " release\n" );
            lastKey = 0;
            shift = 2;
          }
          break;

        case 0xE0:
          // Extended Code
          // This is way oversimplified
          if ( scanCodeBuffer[ 1 ] == 0xF0 )
          // Extended BreakCode
          {
```

```
      getKey( scanCodeBuffer[ index + 2 ] );
      shift = 3;
    }
    else
    // Extended MakeCode
    {
      shift = 2;
    }
    break;

  case CapsLock:
    //Toggle theLEDindicator bit for the correct indicator
    LEDindicators ^= CapsLockLED;
    //Send command to set the LEDs on the keyboard
    dataToSend[ 0 ] = setLEDs;
    dataToSend[ 1 ] = LEDindicators;
    dataToSend[ 2 ] = 0;
    if ( write( fd, dataToSend, 2 ) != 2)
    {
      printf( "sending LED command failed!\n" );
    }
    else
    {
      printf( "LED command sent succesfully!\n" );
    }

    lastKey = CapsLock;
    shift = 1;
    break;

  case NumLock:
    //Toggle theLEDindicator bit for the correct indicator
    LEDindicators ^= NumLockLED;
    //Send command to set the LEDs on the keyboard
    dataToSend[ 0 ] = setLEDs;
    dataToSend[ 1 ] = LEDindicators;
    dataToSend[ 2 ] = 0;
    if ( write( fd, dataToSend, 2 ) != 2)
    {
      printf( "sending LED command failed!\n" );
      }
    else
    {
      printf( "LED command sent succesfully!\n" );
    }

    lastKey = NumLock;
    shift = 1;
    break;

  case ScrollLock:
    //Toggle theLEDindicator bit for the correct indicator
    LEDindicators ^= ScrollLockLED;
    //Send command to set the LEDs on the keyboard
    dataToSend[ 0 ] = setLEDs;
    dataToSend[ 1 ] = LEDindicators;
    dataToSend[ 2 ] = 0;
```

```
            if ( write( fd, dataToSend, 2 ) != 2)
            {
              printf( "sending LED command failed!\n" );
            }
            else
            {
              printf( "LED command sent succesfully!\n" );
            }

            lastKey = ScrollLock;
            shift = 1;
            break;


          default:
            // Normal MakeCode
            thisCode = getKey(scanCodeBuffer[ index ]);
            if( thisCode == LShift || thisCode == RShift )    //Left or
Right Shift
            {
              shift_held = 1;
            }
              if ( scanCodeBuffer[ index ] == 0x09 ) //F10 Key == Exit
              {
                STOP = TRUE;
                printf( "\n\n\t\tGame Over!\n\n" );
              }
              //Handle this part on the breakcode, not the makecode
              else if ( scanCodeBuffer[ index ] == 0x05 ) //F1 Key ==
Toggle encrypting
              {
              if( encryption_on )
                //Encryption is currently on.  It needs to be turned off
              {
                char dataToSend[ 2 ];
                dataToSend[ 0 ] = stop_enc;
                if ( write( fd, dataToSend, 1 ) != 1)
                {
                  printf( "sending Encryption Stop command failed!\n" );
                }
                else
                {
                  printf( "\n\n\t State: G\n\n" );
//                printf( "Stopping Encryption... \n" );

                  encryption_on = 0;
                  K[ 0 ] = Ka0;
                  K[ 1 ] = Ka1;
                  K[ 2 ] = Ka2;
                  K[ 3 ] = Ka3;
                  K[ 4 ] = Ka4;
                  K[ 5 ] = Ka5;
                  K[ 6 ] = Ka6;
                  K[ 7 ] = Ka7;

                  initRC4();
```

```
                getNewKey();
              }
            }
            else
              //Encryption is currently off, it needs to be turned on
            {
              InitiateEncryption( newKey );
            }
            }
            else if ( isLetter( thisCode) )
            {
              if ( ( LEDindicators & CapsLockLED ) || shift_held != 0 )
                  //if Caps Lock is on, convert key to Upper Case.
                  //Eventually fix it so that if Shift key is pressed
                  //the same thing will happen
            {
                thisCode -= 0x20;
            }
              printf( "%c", thisCode );
            }
            else if( isNumber( thisCode ) )
            {
              printf( "%c", thisCode );
            }
            else if ( thisCode == ' ' || thisCode == '\n' || thisCode
== ';' || thisCode == '.'
            || thisCode == ',' || thisCode == '/' || thisCode == '`' ||
thisCode == '='
            || thisCode == '-' || thisCode == '[' || thisCode == ']' ||
thisCode == '\\'
            || thisCode == '\t' )
            {
              printf( "%c", thisCode );
            }
            else if( thisCode == '\b' )
            //The backspace '\b' code just moves the cursor back one
position
            //To make it delete the previous character (like a
backspace), we have to move back one space,
            //print a space character (' '), then move back one space
again
            {
            printf( "\b \b" );
            }
            lastKey = scanCodeBuffer[ index ];
          shift = 1;
      }     //END switch ( scanCodeBuffer[ 0 ] )
      fflush(stdout);

      for( i = 0; i < shift; i++ )
      {
        scanCodeBuffer[ i ] = scanCodeBuffer[ i + 1 ];
      }     //END for( i = 0; i < res - shift; i++ )
      totalBufLen -= shift;
      }           //END while( res > 0 && STOP == TRUE )
      wait_flag = TRUE;
    }       //END if ( wait_flag == FALSE )
```

```
  }         //END while (STOP == FALSE )
  /* restore old port settings */
  //tcsetattr( fd, TCSANOW, &oldtio );
  return 0;
}

unsigned char getKey( unsigned char sCode )
// getKey() returns the ASCII code for a given Scan Code
{
  return scanCode[ sCode ];
}

int initRS232( void )
{
  int fd;
  struct termios oldtio, newtio;
  struct sigaction saio;

  /* open the device to be non-blocking (read will return
automatically) */
//Keyboard Clock = Green/Yellow, PortBbits.RB1
//Keyboard Data = Red/White, PortBbits.RB2
  switch( getComPort() )
  {
      case 1:
            fd = open( COMA, O_RDWR | O_NOCTTY | O_NONBLOCK);
            printf( "Com Port A: /dev/ttyS16 chosen\n" );
            break;
      case 2:
            fd = open( COMB, O_RDWR | O_NOCTTY | O_NONBLOCK);
            printf( "Com Port B: /dev/ttyS17 chosen\n" );
            break;
      case 3:
            fd = open( COMC, O_RDWR | O_NOCTTY | O_NONBLOCK);
            printf( "Com Port C: /dev/ttyUSB0 chosen\n" );
            break;
      case 4:
            fd = open( COMD, O_RDWR | O_NOCTTY | O_NONBLOCK);
            printf( "Com Port B: /dev/ttyUSB1 chosen\n" );
            break;
      default:
            printf("Invalid com port selected.  goodbye.");
            getComPort;
  }

  /* install the signal handler before making the device asynchronous
*/
  saio.sa_handler = signal_handler_IO;
  sigemptyset( &saio.sa_mask );     //saio.sa_mask = 0;
  saio.sa_flags = 0;
  //saio.sa_restorer = NULL;
  sigaction ( SIGIO, &saio, NULL );

  /* allow the process to receive SIGIO */
  fcntl( fd, F_SETOWN, getpid());
  /* Make the file descriptor asyncronous */
  fcntl( fd, F_SETFL, FASYNC );
```

```
  tcgetattr( fd, &oldtio );
  newtio.c_cflag = baudRate | CS8 | CLOCAL | CREAD;
  newtio.c_iflag = IGNPAR;
  newtio.c_oflag = 0;
  newtio.c_lflag &= ~( ICANON | ECHO | ECHOE | ISIG );
  newtio.c_cc[VMIN] = 1;
  newtio.c_cc[VTIME] = 0;
  tcflush( fd, TCIFLUSH );
  tcsetattr( fd, TCSANOW, &newtio );
  return fd;
}

void signal_handler_IO ( int status )
{
  wait_flag = FALSE;
}

void initRC4 ( void )
{
  unsigned char i, j;
  unsigned char tmp;

  //  Initialize T from the key, K
  i = 0;
  for (;;)
  {
    S[i] = i;
    T[i] = K[i % keylength];
    if (i == 255)
    {
      break;
    }
    i++;
  }

  //Permutations of S
  j = 0; i = 0;
  for (;;)
  {
    j = (j + S[i] + T[i]); // %256 isn't needed because this is a
char... it can't be more than 256 anyway
    tmp = S[i]; S[i] = S[j]; S[j] = tmp;
    if (i == 255)
    {
      break;
    }
    i++;
  }

  istream = 0; jstream = 0;

  //  Do over 256 rounds of RC4 as per Ron Rivest's reccomendation --
we don't care about the returned values
  for (i=0; i<255; i++) { cycle(); }
  for (i=0; i<64; i++) { cycle(); }
}
```

```
void cycle ( void )
{
  unsigned char tmp;
  istream = ( istream + 1 ) % 256;
  jstream = (jstream + S[istream] ) % 256;
  tmp= S[istream];
  S[istream] = S[jstream];
  S[jstream] = S[tmp];
}

unsigned char decrypt( unsigned char mcode )
{
  unsigned char tmp;

  cycle();
  tmp = ( S[istream] + S[jstream] ) % 256;
  mcode ^= S[ tmp ];  // %256 isn't needed because this is a char... it
can't be more than 256 anyway
  return mcode;
}

int isLetter( unsigned char asciiByte )
{
  if ( asciiByte >= 0x61 && asciiByte <= 0x7A )
  {
    return 1;
  }
  else
  {
    return 0;
  }
}

int isNumber( unsigned char asciiByte )
{
  if ( asciiByte >= 0x30 && asciiByte <= 0x39 )
  {
    return 1;
  }
  else
  {
    return 0;
  }
}

int getComPort()
{
  int comPort = 0;
  printf("Choose COM Port: \n\t1 or 2\n\n");
  scanf("%d",&comPort);
  if ( comPort > 6 || comPort < 1 )
  {
    printf( "Invalid Entry!" );
    return getComPort();
  }
  return comPort;
```

```
}

void InitiateEncryption( unsigned char* key )
//should be called when BAT is received
{
  //The new key was created towards the beginning of main()
  //Once InitRC4 is called again, the cipher will be intialized with
the new key
  char dataToSend[ 2 ];
  dataToSend[ 0 ] = start_enc;
  if ( write( fd, dataToSend, 1 ) != 1)
  {
    printf( "sending Encryption command failed!\n" );
  }
}

SendKeyChar( int index )
{
  //sleep( 1 );
  char dataToSend[ 3 ] = { 0 };
  dataToSend[ 0 ] = NEXT_KEY_CHAR;
  dataToSend[ 1 ] = index;
  dataToSend[ 2 ] = newKey[ index ];
  if ( write( fd, dataToSend, 3 ) != 3)
  {
    printf( "sending Key Char failed!\n" );
  }
}

getNewKey()
{
  //Generate new key
  long int useconds;
  long int seconds;
  struct timeval tv;
  int k;

  gettimeofday( &tv, NULL );
  useconds = tv.tv_usec;
  seconds = tv.tv_sec;
  srand( useconds );
  for ( k = 0; k < 67; k++ )
  {
    rand();
  }

  //Seed the prng with useconds srand( useconds ).
  //Get a certain number of results from the prng (rand()),
  //then XOR that value with seconds, and use that to seed
  //the prng again, and get enough values to build the key (16 bits per
prng cycle, so 4 cycles)
  //
  //The best thing to do would be to hash the results from the PRNG
with MD5 or another hash function
  srand( rand() ^ seconds );
  k = rand();
  K[ 0 ] = ( unsigned char )( k & 0xFF );
```

```
  K[ 1 ] = ( unsigned char )( ( k >> 8 ) & 0xFF );

  k = rand();
  K[ 2 ] = ( unsigned char )( k & 0xFF );
  K[ 3 ] = ( unsigned char )( ( k >> 8 ) & 0xFF );

  k = rand();
  K[ 4 ] = ( unsigned char )( k & 0xFF );
  K[ 5 ] = ( unsigned char )( ( k >> 8 ) & 0xFF );

  k = rand();
  K[ 6 ] = ( unsigned char )( k & 0xFF );
  K[ 7 ] = ( unsigned char )( ( k >> 8 ) & 0xFF );

  //Done initializing session key
  //
  int i;
  sum = 0;

  for( i=0; i<8; i++ )
  {
    sum += K[ i ];
    newKey[ i ] = decrypt( K[ i ] );
  }

  initRC4();
  //Session Key has now been encrypted with the initial key and stored,
  //and the cipher has been reinitialized with the new key.
  //That way, when it's time to send the new key, we don't have to
worry about
  //doing anything other than sending it and starting to use it as soon
as the PIC is ready
}
```