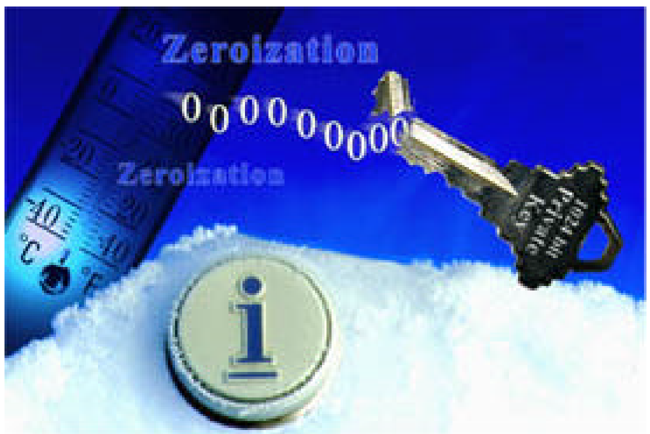
DS1955 Java™-Powered
Cryptographic iButton™



FIPS 140-1 Non-Proprietary
Cryptographic Module Security Policy

Level 3 Validation

January 2000

Table of Contents

1 Introduction

1.1 Purpose

This is a non-proprietary Cryptographic Module Security Policy for the Dallas Semiconductor DS1955 Java™-Powered Cryptographic iButton™ (Java iButton). This security policy was prepared as part of FIPS 140-1 certification of the Java iButton. FIPS 140-1 (Federal Information Processing Standards Publication 140-1 -- *Security Requirements for Cryptographic Modules*) gives U.S. Government requirements for cryptographic modules, and defines the Security Policy as:

> "A precise specification of the security rules under which the cryptographic module must operate, including rules derived from the security requirements of this standard, and the additional security rules imposed by the manufacturer."

The Java iButton provides extraordinary security, meeting all FIPS 140-1 level 3 requirements, and some level 4 requirements. This security policy describes how the Java iButton meets these requirements, and how it can be operated in a secure fashion.

1.2 For more information

This document describes the operations and capabilities of the DS1955 Java™-Powered Cryptographic iButton™ in the technical terms of a FIPS 140-1 cryptographic module security policy.

> For more detailed information about the Java iButton, please visit the iButton web site at http://www.ibutton.com. The web site contains non-technical descriptions of Dallas iButton products, technical specifications, product offerings, iButton functionality, iButton developer information, and much more.
>
> Fore more information about the FIPS 140-1 standard and validation program please visit the NIST web site at http://csrc.nist.gov/cryptval/.
>
> For answers to technical or sales related questions please refer to the contacts listed on the iButton web site at http://www.ibutton.com, or the Dallas Semiconductor web site at http://www.dalsemi.com.

1.3 Terminology

In this document the Dallas Semiconductor DS1955 Java™-Powered Cryptographic iButton™ is referred to as the DS1955, Java iButton (JiB), cryptographic module, Java-Powered Crypto iButton, or module. The JiB is also referred to as simply "iButton", although this term also applies collectively to many other iButtons such as the DS1990, DS1994, or DS1920.

1.4 Document Organization

The Security Policy document is part of the complete FIPS 140-1 Submission Package. In addition to this document, the complete Submission Package contains:

- Vendor Evidence document
- Finite State Machine
- Module Software Listing
- A list of referenced Supporting Documents

This document provides an overview of the Java iButton and explains the secure configuration and operation of the module. This introduction section is followed by Section 2, which details the general features and functionality of the Java iButton. Section 3 specifically addresses the required configuration for the FIPS-mode of operation.

This Security Policy and other Certification Submission Documentation was produced by Corsec Security, Inc. under contract to Dallas Semiconductor. With the exception of this Non-Proprietary Security Policy, the FIPS 140-1 Certification Submission Documentation is Dallas-proprietary and is releasable only under appropriate non-disclosure agreements. For access to these documents, please contact Dallas Semiconductor.

2 The DS1955 Java™-Powered Cryptographic iButton™

The Java™-Powered Cryptographic iButton provides hardware cryptographic services such as a high-speed math accelerator for 1024-bit public key cryptography, and secure message digest (hashing). In FIPS 140-1 terminology, the Crypto iButton is a "multi-chip standalone cryptographic module"; however, the Java iButton actually provides all its services using a single silicon chip packaged in a 16mm stainless steel case. Thus, the iButton can be worn by a person or attached to an object for up-to-date information at the point of use. The steel button is rugged enough to withstand harsh outdoor environments, and is durable enough for a person to wear everyday on a digital accessory like a ring, key fob, wallet, or badge.

Figure 1 – The DS1955 Java™-Powered Cryptographic iButton™ is laser–engraved in steel and silicon

2.1 The iButton Cryptographic Module

The cryptographic boundary for the iButton is the surrounding steel shell. This surrounding shell is factory-lasered with the module's unique 64-bit registration number as shown in Figure 2. The figure shows a button with registration number "1A1D2516"$_{16}$, which is engraved on the encased silicon chip.

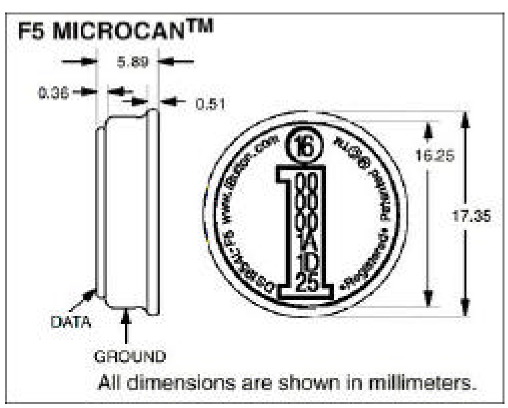


Figure 2 – Java™-Powered Crypto iButton Case and Module Boundary

The ground side of the iButton may optionally be branded with any logo facing. Registration numbers are also lasered into unalterable ROM on the iButtons, which can be read by any application communicating with an iButton. Strict factory controls ensure that registration numbers are globally unique, guaranteeing that no two iButtons ever share a registration number.

2.1.1 Module Interfaces

The button uses a single data contact on the front of the steel case to convey the module's five logical interfaces: data input, data output, control input, status output, and power. These interfaces are logically separated using the 1-Wire™ protocol, which regulates communications and separates reading, writing, and power applied to the module. The 1-Wire protocol utilizes a scratchpad buffer and features atomic, packetized transfers which assures error-free transmission, even with an intermittent connection, in addition to complete separation of input, processing, and output phases. Control commands and data must be input in error checked packets, and data and status are returned only after successful completion of processing.

2.1.2 Module Components

The active components of the iButton are shown below, and consist of a lithium cell (for backup power), an energy reservoir (to provide parasitic capacitance power), a quartz timing crystal (for a True Time Clock), and the single DS83C960 cryptographic chip.

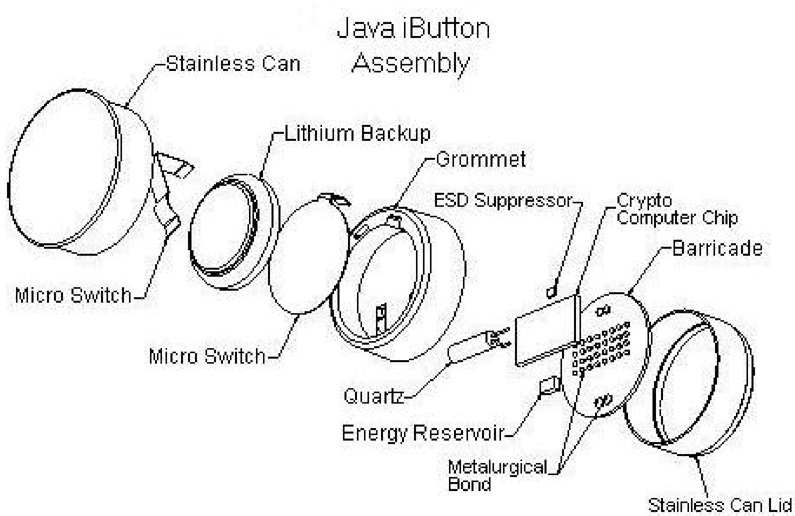


Figure 3 – Components of the Crypto iButton™

2.2 Physical Security

The Java™-Powered Crypto iButton boasts an incredible array of physical security safeguards packed into a small coin-sized device. Because the silicon chip is encased in stainless steel, the iButton will stand up to the harsh conditions of daily wear, including dropping it, stepping on it, and inadvertently passing it through the washing machine and dryer.

Figure 3 – The Crypto iButton Mounted as a Signet Jewel of a Ring.
The Java iButton an be Attached to any Personal Accessory.

2.2.1 The Strength of Steel

The tough stainless steel case of the iButton defines a contiguous perimeter and provides clear visual evidence of tampering. The module does not contain any holes or vents that could permit probing. Tamper-signs include mangling and scratching of the data and ground plates and the smooth grommet separation. It is this outer case which satisfies FIPS 140-1 tamper evidence requirements for physical security. In addition, the case meets the FIPS 140-1 level 3 tamper response and level 4 Environmental Failure Protection (EFP) requirements.

2.2.2 Goes Down in a Blaze of Zeroization

If an iButton is pried open, a microswitch triggers an active zeroization of the chip's contents, destroying private keys and other sensitive information. The iButton constantly monitors the switch's contacts, and any separation of the cryptographic chip from the lithium cell switches the device to on-chip capacitor power to perform a complete zeroization as its last powered action.

2.2.3 Neither snow nor rain nor heat…[1]

Orchestrated attacks to uncover iButton secrets by subjecting the iButton to extreme temperature or voltage conditions will generate a tamper response that results in zeroization. Deliberately exposure to temperatures outside the iButton's operational range of -20^{o}C to +70^{o}C (-4^{o}F to +158^{o}F) cause temperature monitors to trigger a cold-

[1] "*Neither snow nor rain nor heat nor gloom of night stays these couriers from the swift completion of their appointed rounds.*" -- an inscription on the General Post Office, New York City. (see http://www.usps.gov/history/his8.htm)

temp switch or high-temp effects that quickly zeroize to erase the contents of the memory. Voltages above or below maximum operating tolerances are clamped, and if excessive voltage is encountered, the I/O pin is designed to fuse and render the chip inoperable.

2.2.4 Fortresses large and microscopic…

In addition to these operation controls, the cryptographic chip is additionally protected by a substrate barricade. A substrate barricade is metallurgically- and glass epoxy-bonded to the active face of the chip. Attempts to remove the barrier to get to the chip cause a tamper response that results in zeroization. If a sophisticated attacker attempts to micro-probe the chip, they will encounter a shield of sub-micron pitch metal layers fabricated into a serpentine pattern directly on the chip. The chip will detect any break in this shield and immediately zeroize the chip.

2.3 DS1955 Firmware Capabilities

The Java iButton firmware, which includes a Java virtual machine, runs on a single, state-of-the-art silicon chip. The Java iButton contains:

- an 8051-compatible microcontroller,
- a protected real-time clock,
- a high-speed modular exponentiation accelerator for large integers up to 1024 bits in length,
- 64 Kbytes of ROM memory with preprogrammed firmware,
- 6 Kbytes of non-volatile RAM (NVRAM) for storage of critical data,
- input and output buffers with the standard iButton 1-Wire "front-end" for sending and receiving data, and control circuitry that enables the microcontroller to be powered up to interpret and act on the data placed in an input buffer, drawing its operating power from the 1-Wire line.

The microcontroller, clock, memory, buffers, 1-Wire front-end, modular exponentiation accelerator, and control circuitry are integrated on a single silicon chip and packaged in a stainless steel case using packaging techniques which make it virtually impossible to probe the data in the NVRAM without destroying the data. Most of the NVRAM is available for use to support cryptographic applications such as those mentioned above.

The Java iButton firmware supports the Secure Hashing Algorithm (SHA-1) and conforms to Federal Information Processing Standard Publication (FIPS PUB) 180-1, *Secure Hash Standard (SHS).*

2.4 Roles & Services

There are two separate roles in the operation of Java iButtons: Crypto Officer, and User. The Java™-Powered Crypto iButton is intended to be activated by the factory (Crypto

Officer). Registered customers (Users) operate the devices as end-users under a license agreement. The Crypto Officer loads the Java™-Powered Crypto iButton with data to enable it to perform application-specific functions. The User issues commands to the Java iButton to perform operations programmed by the Crypto Officer. For this reason the Java iButton offers functions to support the Crypto Officer in setting up the Java iButton for an intended application, and it also offers functions to allow the authorized User to invoke the services offered by the Crypto Officer.

2.4.1 Authentication

The Java iButton™ provides identification and authentication (I&A) functions for both role-based and identity-based I&A.

2.4.1.1 Identity-Based Authentication

The Java iButton™ can perform identity-based authentication using challenge-response protocols that use keyed message digest algorithms. Using these protocols, the iButton™ allows users to log in and log out. When operated in the FIPS mode, the Java iButton™ employs identity-based authentication using challenge-response protocols. This advanced I&A, may be used in conjunction with the role-based authentication described below.

2.4.1.2 Role-Based Authentication

The Java iButton™ may optionally use role-based authentication, in addition to or in lieu of identity-based I&A. There is a Crypto Officer personal identification number (PIN) for each iButton, which must be supplied with each and every service request reserved for the Crypto Officer. The Crypto Officer PIN (also called the iButton's common PIN) can be any value (numeric, alpha, or binary byte values), and is eight bytes in length. Similarly, there is a User PIN, which must be supplied with every request for User services. There are also non-cryptographic services (related to iButton status) which are available to User and Crypto Officer without supplying an authenticating PIN.

2.4.2 Crypto Officer Services

A Crypto Officer can exercise the following services with appropriate authentication:

Master Erase – The entire contents of the RAM is zeroized. This will delete any applets that are loaded, clear any existing logon statuses, and re-initialize the module to factory defaults.

Set Master PIN – This function enables the Crypto Officer to enter the master PIN number that will be used to access the module.

Set Ephemeral GC Mode - The ephemeral collector recovers data that was referenced for a short period of time and then went out of scope (Objects whose references were never stored in reference fields, etc).

Set Applet GC Mode - The applet collector recovers data that was referenced by the fields of an applet. These references were new-ed and then the references were lost either by setting the field to null or by new-ing another block of data.

Set Command PIN Mode - When Command PIN Mode is enabled, all commands require a PIN match before the command is executed. When disabled, the PIN match is skipped.
Set Load PIN Mode - When Load PIN Mode is enabled, nine bytes of PIN data, pre-pended to the applet data (JiB file data), must match the Master PIN in order for the applet load to succeed. When disabled, the applet must be signed
Set Restore Mode - When Restore Mode is enabled, All field updates and system transactions are considered atomic. If a tear occurs in the middle of these updates, the values just prior to the update are restored.
Set Exception Mode - When Exception Mode is enabled, java API exceptions are thrown. All uncaught exceptions return 0x6f00 in the SW. When disabled, an error is returned from the Virtual Machine (VM).
Set Commit Buffer Size - Committing one field to the buffer requires 9 bytes. Therefore the default size of 72 bytes allows 8 field updates. The minimum size allowed is 72 bytes and the maximum is restricted by the amount of free RAM. All values will be rounded up to the next multiple of 9.
Run Self-Tests **–** initiates the running of the FIPS-required self-tests, specifically, the SHA-1 test and the Statistical random number generator test. This service returns either a status of either success or failure of the tests.

These functions allow the Crypto Officer to completely erase and zeroize an iButton, add new applets, and lock the iButton to prevent additional applets from being added or changed.

A complete description of each Java iButton command can be found in FIPS submission Document 1O, iButton Commands Reference.

2.4.3 User Services

A User, by default, cannot execute any functions other than the status functions.

2.4.4 Status Functions

A number of status functions can be used to find the state of the iButton and various configuration information about the iButton. These status functions can be used by both User and Crypto officer without supplying any PIN:

Get Firmware Version String **–** Returns the Firmware Version String (FVS) in the following format: [length byte - Len][Len bytes of FVS data].
Get Free RAM - Returns a short (least significant block (LSB) first) representing the amount of free RAM remaining in the iButton.
Get Ephemeral GC Mode - Returns a byte representing the mode – 0 for disabled, 1 for enabled.
Get Applet GC Mode - Returns a byte representing the mode – 0 for disabled, 1 for enabled.

Get Command PIN Mode - Returns a byte representing the mode – 0 for disabled, 1 for enabled.
Get Load PIN Mode - Returns a byte representing the mode – 0 for disabled, 1 for enabled.
Get Restore Mode - Returns a byte representing the mode – 0 for disabled, 1 for enabled.
Get Exception Mode - Returns a byte representing the mode – 0 for disabled, 1 for enabled.
Get Commit Buffer Size - Returns a short (LSB first) representing the size, in bytes, of the Commit Buffer.
Get Real Time Clock - Returns a 4 byte number (LSB first) representing the current value of the real-time clock in seconds. This value counts up from zero and represents the amount of time since the battery was attached.
Get Random Bytes - Input data: Short value (LSB first) representing the number of random bytes to retrieve.
List Applet by Number - Returns the applet identification (AID), or applet name, in the format [AID length - Len][Len bytes of AID data].
Get POR Count - Returns a short (LSB first) representing the number of Power On Resets that have occurred since the last Master Erase.
Get State – Returns the current state the module is in as status information. The state number is returned.

2.5 Key Management

No keys are implemented in the iButton. The Java iButton has a unique internal 64-bit registration number which is not a key, is not private, and is also engraved on the outside of the module.

3 Java™ iButton™ FIPS Mode

The Java™ iButton™ provides a rich set of cryptographic functionality in a physically secure package. This versatile module can be configured to function in a wide variety of applications such as an electronic change purse, biometric access token, or postage meter. The iButton™ can also be configured to operate in a FIPS 140-1 compliant mode. When configured and operated in this mode, the Java™ iButton™ provides a FIPS 140-1 level 3 compliant cryptographic module.

In all forms of operation, the Java™ iButton™ meets most level 3 and some level 4 FIPS 140-1 physical security requirements using sophisticated hardware security controls. Other areas of FIPS level 3 requirements can only be met if the Java™ iButton™ is configured for and operated in its FIPS mode.

3.1 FIPS Restrictions

FIPS 140-1 requires the use of FIPS approved algorithms and does not allow the use of RSA public key algorithms for encryption or decryption of data, nor does it currently allow the industry-standard PKCS #1 version of RSA digital signatures. Therefore, the Java™-Powered Crypto iButton™ uses only SHA-1 for message digests in the FIPS mode.

The Java™-Powered Crypto iButton™ provides several methods of identification and authentication to provide role-based or identity-based authentication. In order to meet level 3 FIPS 140-1 requirements the iButton™ must be operated using identity-based authentication. The module is operated as a single-user device with identity-based authentication of the user as described in section 2.4.1.1. The authentication used in the FIPS mode incorporates a challenge-response protocol for login. The protocol ensures that no plaintext authentication data is transmitted over the 1-Wire™ interface. In addition, this authentication does not use RSA encryption, instead utilizing a keyed message digest algorithm incorporating SHA-1.

For operation in FIPS mode, no plaintext keys may be exported from the iButton™. Therefore, all keys are privatized and only accessed internally by applets.

3.2 FIPS Configuration

To configure the iButton™ for operation in FIPS mode, the factory performs the following operations:

- Master Erase the iButton™, removing all previous data.
- Initialize the common PIN to a random value known only to the factory. The common PIN is set with the "`Set Master PIN`" command.
- Load and install the FIPS 140-1 Applet.
- Set a User login password.
- Lock the iButton to prevent loading additional applets, or access to any administrative commands.

- Set the seed value.
- Deliver the FIPS mode iButton™ and User login information.

Listed here are all the iButton modes set at the factory and what are set to before shipping:

- Master Pin: Set to a Random Non-Released value
- Applet GC Mode 1
- Command PIN Mode 1 (default)
- Load PIN Mode 1 (default)
- Restore Mode 1 (default)
- Exception Mode 1 (default)
- Commit Buffer Size 72 (default)

3.3 Operation in FIPS mode

Operation of a Java iButton™ in FIPS mode is limited to functions available from the FIPS applet. Because the factory initializes the button and does not release the Common PIN for the any FIPS mode iButton™, the user cannot exercise any of the services listed in section 2.4.2.

The User may not call any services other than the status services, however, the User can also invoke one of the five FIPS applet functions, Get Challenge, Login, Logout, Run Self Tests, and SHA-1 Hash. The User must first login to the iButton™ using the Login script. To do this, the user calls the Get Challenge function to receive the random challenge data, and computes the SHA-1 hash of the random challenge appended to the User login password. This response is provided to the Login script.

Once a User has logged in, he may run the SHA-1 Digest script to hash data, or run the Logout script. Once a User has been erased, the User can no longer login and the FIPS mode iButton™ must be returned to the factory.

The User may also run the Self Tests from any of the user states. If the module fails any part of the Self Tests, the module transitions to an error state that restricts the available services.

3.4 *Factory Configuration Reference*

All FIPS mode iButtons™ are delivered from the factory tested, operational, and configured. The applet described below, which contains the FIP-mode functions, is loaded into the iButton™ customized with a User login password. The full factory configuration process is described in section 3.2.

```
//FIPS 140-1 Applet for the FIPS 140-1 Level-3 Compliant configuration
//of the DS1955 Java-Powered iButton.
//
//Modifications:  Initial version -- Corsec Security, Inc. Dec. 1999

import javacard.framework.*;
import javacardx.crypto.*; //For cryptographic functions like SHA-1

public class FIPS_140_1_Applet extends Applet
{
    // BEGIN INSTRUCTION DECLARATIONS
    public static final byte FIPS_140_1_CLA = (byte)0x80;

    public static final byte FIPS_140_1_INS_LOGIN = (byte)0;
    public static final byte FIPS_140_1_INS_LOGOUT = (byte)1;
    public static final byte FIPS_140_1_INS_GETCHALLENGE = (byte)2;
    public static final byte FIPS_140_1_INS_SETSEED = (byte)3;
    public static final byte FIPS_140_1_INS_SHA1HASH = (byte)4;
    public static final byte FIPS_140_1_INS_SETPASS = (byte)5;
    public static final byte FIPS_140_1_INS_GETSTATE = (byte)6;
    public static final byte FIPS_140_1_INS_SELFTEST = (byte)7;
    //   END INSTRUCTION DECLARATIONS

    //State variable enumerations
    public static final short STATE_POST = (short)0;  //Power up self test state
    public static final short STATE_POST_FAIL = (short)7;  //POST Self tests have failed
    public static final short STATE_NOPASS = (short)8; //No password set yet
    public static final short STATE_UNSEEDED = (short)9; //No seed set yet
    public static final short STATE_LOGGEDOUT = (short)10; //Not logged in
    public static final short STATE_LOGGEDIN = (short)11; //Logged in
    public static final short STATE_RUN_FAIL = (short)12;  //During run,Self tests failed

    //The parameters used to select an applet.
    final static private byte SELECT_CLA = (byte)0x00;
    final static private byte SELECT_INS = (byte)0xA4;

    //The Maximum number of bytes we will send in a single apdu.
    final static private short MAX_SEND_LENGTH = (short)1000;

    //BEGIN GLOBAL VARIABLES
    private int state; //The machine state
    //User Password
    private byte[] password= new byte[8];
    private int passlength;  //the length of the User Password.
    private byte[] randomChallenge=new byte[20]; //Random data used for challenge response
    private byte[] lastRandom=new byte[20];  //last Random number generated.
                                             //(Stored for continuous RNG test.)
    public byte[] apduData; //Application Protocol Data Unit Data (input/output for Applet)
    private RandomData randGenerator=new RandomData(); //Random number generator object
    //END GLOBAL VARIABLES


    public FIPS_140_1_Applet() //Constructor called after install.
    {
        //Register this applet with the JCRE
        register();

        //Initialize state value to power-up self test state
        state=STATE_POST;

        //perform SHA-1 known answer test
        if (SHA_1_KAT()) //if we get the expected SHA result
```

```java
            state=STATE_NOPASS; //mark the state to no password
        else
            state=STATE_POST_FAIL; //otherwise we failed self-tests
    }

    //Install function is called when the applet is first loaded.  It is
    // then stored in memory and ready to be selected and then processed.
    public static void install(APDU apdu)
    {
        new FIPS_140_1_Applet(); //Create an instance and run constructor
    }

    public void process(APDU apdu)
    {
        //This function is what is called when the applet is run by
        // the Java Virtual Machine.  It reads the APDU, makes sense of it,
        // and calls the various dispatcher functions (which execute based on
        // the received INS value)..
        byte[] buffer = apdu.getBuffer();

        //Determine if the applet is being selected.
        if((buffer[ISO.OFFSET_CLA] == SELECT_CLA) &&
           (buffer[ISO.OFFSET_INS] == SELECT_INS))
        {
            return;
        }

        apduData = new byte[buffer[ISO.OFFSET_LC] & 0x0FF];
        short apduDataOffset = 0;

        //Read in the entire APDU.
        short bytesRead = apdu.setIncomingAndReceive();
        //Loop until all bytes have been read.
        while (bytesRead > 0)
        {
            Util.arrayCopyNonAtomic(buffer, ISO.OFFSET_CDATA, apduData,
                 apduDataOffset, bytesRead);
            apduDataOffset += bytesRead;
            bytesRead = apdu.receiveBytes(ISO.OFFSET_CDATA);
        }
        //Prepare the apdu for sending.
        apdu.setOutgoing();
        apdu.setOutgoingLength(MAX_SEND_LENGTH);

        //Check for a valid CLA.
        if(buffer[ISO.OFFSET_CLA] != FIPS_140_1_CLA)
        {
            ISOException.throwIt(ISO.SW_CLA_NOT_SUPPORTED);
        }
        else
        {
            //Call the appropriate dispatch method for the given INS.
            switch (buffer[ISO.OFFSET_INS])
            {
                case FIPS_140_1_INS_LOGOUT:
                    logoutDispatch(apdu, buffer[ISO.OFFSET_P1], buffer[ISO.OFFSET_P2]);
                    break;
                case FIPS_140_1_INS_GETCHALLENGE:
                    getchallengeDispatch(apdu, buffer[ISO.OFFSET_P1], buffer[ISO.OFFSET_P2]);
                    break;
                case FIPS_140_1_INS_SETSEED:
                    setseedDispatch(apdu, buffer[ISO.OFFSET_P1], buffer[ISO.OFFSET_P2]);
                    break;
                case FIPS_140_1_INS_SHA1HASH:
                    sha1hashDispatch(apdu, buffer[ISO.OFFSET_P1], buffer[ISO.OFFSET_P2]);
                    break;
                case FIPS_140_1_INS_SETPASS:
                    setpassDispatch(apdu, buffer[ISO.OFFSET_P1], buffer[ISO.OFFSET_P2]);
                    break;
                case FIPS_140_1_INS_GETSTATE:
                    getstateDispatch(apdu, buffer[ISO.OFFSET_P1], buffer[ISO.OFFSET_P2]);
                    break;
                case FIPS_140_1_INS_SELFTEST:
                    selftestDispatch(apdu, buffer[ISO.OFFSET_P1], buffer[ISO.OFFSET_P2]);
                    break;
                case FIPS_140_1_INS_LOGIN:
```

```java
                    loginDispatch(apdu, buffer[ISO.OFFSET_P1], buffer[ISO.OFFSET_P2]);
                    break;

                default:
                    ISOException.throwIt(ISO.SW_INS_NOT_SUPPORTED);
            }
        }
    }

// BEGIN INSTRUCTION DISPATCHER FUNCTIONS
//These functions are the ones that actually execute the instructions
//The first logs in, the second logs out, the third returns a random
// challenge, the fourth sets the seed, the fifth returns a SHA1Hash
// the sixth sets the password, and seventh returns the state, and
// the eighth runs the self-tests.

//Login Dispatch- Takes in "login parameters" and if correct transitions
//to login state (provided it is already in the loggedout state).
//Otherwise, it returns an error message of some sort
//Assumes that input data is 20 bytes long
 public void loginDispatch(APDU apdu, byte p1, byte p2)
 {
    if(state==STATE_LOGGEDOUT)//If currently in logged out state
     {
         byte[] hashResult= new byte[20]; //Buffer for SHA hash value
         byte[] inData = new byte[apduData.length]; //Buffer for apdu
         //convert apdu data to byte[] for easier use
         Util.arrayCopyNonAtomic(apduData, (short)0, inData, (short)0,
               (short) apduData.length);
         //initialize SHA-1 Digest Generator
         Sha1MessageDigest digestGenerator=new Sha1MessageDigest();
         //Generate SHA-1 message digest
         digestGenerator.generateDigest(addByteArray(), (short)0, (short)20, hashResult, (short)0);
         //reset random challenge(to prevent multiple attempts on the same challenge)
         if (!getRandom(randomChallenge,1))
         {
             error2(apdu,p1,p2); //If continuous RNG test fails, error message
             return;
         }
         if(compareArray(hashResult, inData)) //If the hash result and the inData are the same.
         {
             state=STATE_LOGGEDIN;//Move to logged in state
             //On success send "Login Passed"
             byte[] tempdata = {(byte)'L',(byte)'o',(byte)'g',(byte)'i',(byte)'n',(byte)' ',
               (byte)'P',(byte)'a',(byte)'s',(byte)'s',(byte)'e',(byte)'d',(byte)0x00};
             sendByteArray(apdu, tempdata);
         }
         else //invalid hash indicates login failure (maybe wrong password)
         {   // On failure send "Login Failed"
             byte[] tempdata = {(byte)'L',(byte)'o',(byte)'g',(byte)'i',(byte)'n',
               (byte)' ',(byte)'F',(byte)'a',(byte)'i',(byte)'l',(byte)'e',
               (byte)'d',(byte)0x00};
             sendByteArray(apdu, tempdata);
         }
     }
     else
         error1(apdu,p1,p2); //If invalid state send error message
 }

 //Logout Dispatch- No input required.  Logs out from logged in state or
 // if in another state returns an error message.
 public void logoutDispatch(APDU apdu, byte p1, byte p2)
 {
     if(state==STATE_LOGGEDIN)//if currently logged in
     {
         state=STATE_LOGGEDOUT; //set the State to logged out.
         //On Success send "Logged Out"
         byte[] tempdata = {(byte)'L',(byte)'o',(byte)'g',(byte)'g',(byte)'e',
             (byte)'d',(byte)' ',(byte)'O',(byte)'u',(byte)'t',(byte)0x00};
         sendByteArray(apdu, tempdata);
     }
     else error1(apdu, p1, p2); //Send invalid state error message
 }

 //Get Challenge Dispatch- If currently in Logged Out State, IE, random
 // number generator seeded, but not logged in, returns a random challenge,
```

```
// otherwise returns an error message.  No input required.
public void getchallengeDispatch(APDU apdu, byte p1, byte p2)
{
    if(state==STATE_LOGGEDOUT)//if logged out but random number generator seeded
    {
        state=STATE_LOGGEDOUT; //Make sure state stays loggedout.
        if (!getRandom(randomChallenge,1)) //generate random challenge
        {
            error2(apdu,p1,p2); //If continuous RNG test fails, error message
            return;
        }
        sendByteArray(apdu, randomChallenge);//send random challenge to user
    }
    else
        error1(apdu, p1,p2); //Send error message "Invalid State"
}

// Set Seed Dispatch- Takes in the seed value, and seeds
// the random number generator.  If the state is currently
// the unseeded state, it transitions it to the logged out state
public void setseedDispatch(APDU apdu, byte p1, byte p2)
{
    byte[] outData = new byte[apduData.length];//output data space

    if (state==STATE_UNSEEDED) //If applet is unseeded still
    {
        state=STATE_LOGGEDOUT; //transition to logged out state
        //convert apdu data to byte[] to send to randGenerator
        Util.arrayCopyNonAtomic(apduData, (short)0, outData,
        (short)0,(short) apduData.length);
        //actually set the seed value.
        randGenerator.setSeed(outData, (short)0, (short)apduData.length);
        //On Success send "Seed Set"
        byte[] tempdata = {(byte)'S',(byte)'e',(byte)'e',(byte)'d',(byte)' ',
           (byte)'S',(byte)'e',(byte)'t',(byte)0x00};
        sendByteArray(apdu, tempdata);
    }
    else error1(apdu, p1, p2); //all other states get error message
}

//SHA-1 Hash Dispatch -- hashes data for a logged in user.
// using FIPS 180-1 (SHA-1) message digesting.
public void sha1hashDispatch(APDU apdu, byte p1, byte p2)
{
    if(state==STATE_LOGGEDIN)//If currently logged in
    {
        byte[] tempData=new byte[20]; //temp data for hashing.
        byte[] inData = new byte[apduData.length]; //copy of apduData as byte

        //convert apdu data to byte[] for easier use
        Util.arrayCopyNonAtomic(apduData, (short)0, inData,
                            (short)0,(short) apduData.length);
        //create hash object
        Sha1MessageDigest digestGenerator=new Sha1MessageDigest();
        //Actually hash the data
        digestGenerator.generateDigest(inData, (short)0,
                     (short)apduData.length, tempData, (short)0);
        //And return it to the user
        sendByteArray(apdu, tempData);
    }
    else
        error1(apdu,p1,p2);//Send invalid state error message
}

//function to set password during initialization
public void setpassDispatch(APDU apdu, byte p1, byte p2)
{
    if(state==STATE_NOPASS)//works only if no pass has already been set
    {
        byte[] newPass= new byte[apduData.length]; //Password value to set
        //convert apdu data to byte[] for easier use
        Util.arrayCopyNonAtomic(apduData, (short)0, newPass,
           (short)0,(short) apduData.length);
        if(apduData.length>8)//check if password is too long
        {  //If password too long, send, "Password Too Long".
            byte[] tempdata = {(byte)'P',(byte)'a',(byte)'s',(byte)'s',(byte)'w',
```

```
                (byte)'o',(byte)'r',(byte)'d',(byte)' ',(byte)'T',(byte)'o',
                (byte)'o',(byte)' ',(byte)'L',(byte)'o',(byte)'n',(byte)'g',(byte)0x00};
            sendByteArray(apdu, tempdata);
        }
        else
        { //set password to the new password
            Util.arrayCopyNonAtomic(newPass,(short)0, password, (short)0, (short)apduData.length);
            passlength=apduData.length; //Change passlength to match new passwords length
            state=STATE_UNSEEDED;       //transition to unseeded state
            //On Success send "Password Set"
            byte[] tempdata = {(byte)'P',(byte)'a',(byte)'s',(byte)'s',(byte)'w',
                (byte)'o',(byte)'r',(byte)'d',(byte)' ',
                (byte)'S',(byte)'e',(byte)'t',(byte)0x00};
            sendByteArray(apdu, tempdata);
        }
    }
    else error1(apdu, p1, p2);  //Invalid state error
}

//return the current applet state (run from any state)
public void getstateDispatch(APDU apdu, byte p1, byte p2)
{
    byte[] outgoingData = new byte[4]; //reserve space for returned state
    intToByteArray(outgoingData, 0, state); //convert state to byte for return
    sendByteArray(apdu, outgoingData);
}

// Self Test Dispatch- runs all the self tests and
// reset the state if they pass.
public void selftestDispatch(APDU apdu, byte p1, byte p2)
{
    if ( (state==STATE_POST_FAIL) ||  //works only in operational or failed states
         (state==STATE_RUN_FAIL) ||  //Available as a service from these states
         (state==STATE_LOGGEDOUT) || //But failure will transition to failed
         (state==STATE_LOGGEDIN) ) //State and require re-run of tests
    {
        boolean TestsPass = false; //flag for whether SHA_1_KAT passes
        int code = 0; //return value from SRNG tests

        TestsPass = SHA_1_KAT(); //Run SHA-1 KAT
        //Run Statistical RNG tests if previous test passed
        if (TestsPass) code = SRNG_Test();
        else code = 7; //code for SHA-1 KAT failure
        if (0 != code) TestsPass = false; //Nonzero return is an error

        if (TestsPass) //If all the self tests have passed.
        {
            //logout if we got here from a running state
            if (state==STATE_RUN_FAIL) state=STATE_LOGGEDOUT;
            //else if we came from POST, go to not initalized state
            else if (state==STATE_POST_FAIL) state=STATE_NOPASS;
            //else for STATE_LOGGEDIN, STATE_LOGGEDOUT,
            // we need no change the self tests were run and passed.
            //Tell the user that the Tests Passed.
            byte[] tempdata = {(byte)'T',(byte)'e',(byte)'s',(byte)'t',(byte)'s',(byte)' ',
              (byte)'P',(byte)'a',(byte)'s',(byte)'s',(byte)'e',(byte)'d',(byte)0x00};
            sendByteArray(apdu, tempdata);
        }
        else //self tests failed.
        {
            if ((state==STATE_LOGGEDIN) || (state==STATE_LOGGEDOUT) || (state==STATE_RUN_FAIL))
                state=STATE_RUN_FAIL; //if were were operational before, go to run failure
            else
                state=STATE_POST_FAIL; //if not, go to initialize restart.
            //choose return message based on code
            if (5==code)
              error2(apdu, p1, p2);  //send error message for failed runs tests
            else if (4==code)
              error2(apdu, p1, p2);  //send error message for failed poker tests
            else if (3==code)
              error2(apdu, p1, p2);  //send error message for failed long runs test
            else if (6==code)
              error2(apdu, p1, p2);  //failed continuous RNG test
            else if (7==code)
              error2(apdu, p1, p2);  //failed SHA-1 KAT test
            else //monobit failure
```

```
            error2(apdu, p1, p2);  //send error message for failed self tests
        }
    }//end if state is correct
    else //called from non-initialized state
        error1(apdu, p1, p2);  //send invalid state error message
}

//   END INSTRUCTION DISPATCHER FUNCTIONS

// BEGIN CONVENIENCE FUNCTIONS
public static void intToByteArray(byte[] outArray, int start, int value)
{ //Convert from integers to Bytes
    //BigEndian
    outArray[start]     = (byte)((value & 0xFF000000) >>> 24);
    outArray[start + 1] = (byte)((value & 0x00FF0000) >>> 16);
    outArray[start + 2] = (byte)((value & 0x0000FF00) >>>  8);
    outArray[start + 3] = (byte) (value & 0x000000FF);
}

protected void sendByteArray(APDU apdu, byte[] data)
{ //Send an array of bytes back to the user
    short offset = 0;
    while((data.length - offset) > MAX_SEND_LENGTH)
    {
        apdu.sendBytesLong(data, offset, MAX_SEND_LENGTH);
        offset += MAX_SEND_LENGTH;
    }
    apdu.sendBytesLong(data, offset, (short)(data.length - offset));
}

private boolean SHA_1_KAT() //Perform SHA-1 Known Answer Test
{   //Known answer taken from the SHA-1 standard examples (FIPS 180-1)
    byte[] answer=new byte[20]; //the result of our sha test
    byte[] knownAnswer= {(byte)0x36,(byte)0x3E,(byte)0x99,(byte)0xA9,
                        (byte)0x6A,(byte)0x81,(byte)0x06,(byte)0x47,
                        (byte)0x71,(byte)0x25,(byte)0x3E,(byte)0xBA,
                        (byte)0x6C,(byte)0xC2,(byte)0x50,(byte)0x78,
                        (byte)0x9D,(byte)0xD8,(byte)0xD0,(byte)0x9C};
                        //the expected result of our SHA test
    byte[] input = { (byte)'a',(byte)'b',(byte)'c'}; //value to be hashed
    Sha1MessageDigest testSHA=new Sha1MessageDigest(); //create sha object
    testSHA.generateDigest(input, (short)0, (short)3, answer, (short)0); //hash
    if(compareArray(answer , knownAnswer)) //if we get the expected SHA result
        return (true);
    else //we have failed the SHA KAT and must go to non-functional state
        return (false);
}
//compare two arrays(both hashes of length 20)
public boolean compareArray(byte[] array1, byte[] array2)
{
    for(int x=0; x<20; x++) //loop through all 20 bytes
    { //and complain if any are different
        if(array1[x]!=array2[x])return(false);
    }
    return(true); //otherwise they are equal
}

// Random number generation for bytes * 20 random data
// Includes a continuous random number generator test
//   returns true if it succeeds
//   returns false upon failure and changes state to STATE_FAIL
private boolean getRandom(byte[] RandomData, int bytes)
{
    byte[] new_Bytes=new byte[20]; //a twenty byte sample value

    for (int y=0; y<bytes; y++) //get (bytes) values each 20 bytes long.
    {
        //get some random data
        randGenerator.generateData(new_Bytes, (short)(0), (short)20);

        //check this data
        if (compareArray(new_Bytes, lastRandom)) //If we got the same value
        {   //NIST requires a failure be reported for this
            state=STATE_RUN_FAIL; //mark it bad and return
            return false; //return failure code
        }
```

```
        else //random value is good
        {   //Record the new value in our global storage for later use
            Util.arrayCopyNonAtomic(new_Bytes, (short)0, lastRandom,
                                    (short)0,(short) 20);
            //save the new random data for output
            Util.arrayCopyNonAtomic(new_Bytes, (short)0, RandomData,
                                    (short)(y*20),(short) 20);
        }
    } //Otherwise we got 20 * bytes of good random data. return success
    return true;
}

//Function to quickly splice the random data together with the pass
private byte[] addByteArray()
{
    byte[] addedValue= new byte[20]; //Spliced value
    Util.arrayCopyNonAtomic(password, (short)0, addedValue,(short)0,
       (short)passlength);//copy password to target(to be returned)
    //copy rest of random data to target
    Util.arrayCopyNonAtomic(randomChallenge, (short)passlength, addedValue,
       (short)passlength, (short)(20-passlength));
    return(addedValue); //return the spliced value
}
//Error message return function
private void error1(APDU apdu, byte p1, byte p2)//returns invalid state error message
{
    byte[] tempdata = {(byte)'I',(byte)'n',(byte)'v',(byte)'a',(byte)'l',(byte)'i',
     (byte)'d',(byte)' ',(byte)'S',(byte)'t',(byte)'a',(byte)'t',(byte)'e',(byte)0x00};
    sendByteArray(apdu, tempdata); //Send invalid state error message
}
//Error message return function
private void error2(APDU apdu, byte p1, byte p2)//returns self test fail error message
{
    byte[] tempdata = {(byte)'S',(byte)'e',(byte)'l',(byte)'f',(byte)' ',(byte)'T',
     (byte)'e',(byte)'s',(byte)'t',(byte)' ',(byte)'F',(byte)'a',(byte)'i',(byte)'l',
     (byte)0x00};
    sendByteArray(apdu, tempdata); //Send "Self Test Fail" message
}

//Implement the FIPS 140-1 Statistical Random Number Generation Tests (SRNG)
//Runs the tests and returns true on success or false if anything fails.
//
// (Actually this implements FIPS 140-2 SRNG tests, which are the same except)
// (the acceptable values from the RNG must be closer to norms as noted)
// (in the comments below)
//The SRNG Tests have four parts that operate on a 20,000 bit random stream:
//  Monobit Test
//         Count the number of ones
//         Must be between 9,654 and 10,346 (FIPS 140-2: 9,725 and 10,275)
//  Poker Test
//         Take every four bits (5000 samples)
//         Record how many of samples fall into each of sixteen four bit possibilities
//         Sum the squares of the sixteen values = sigma
//         16/5000 * sigma - 5000 = X
//         X between 1.03 and 57.4 (FIPS 140-2 X between 2.16 and 46.17)
//  Runs Test
//         Count runs of zeros or ones from one to six
//         Must be in following totals for all twelve counts:
//      Run       FIPS 140-1 Range      FIPS 140-2 Range
//         1       2267-2733                           2,315-2,685
//         2       1079-1421                           1,114-1,386
//         3       502-748                             527-723
//         4       223-402                             240-384
//         5       90-223                              103-209
//         6+      90-223                              103-209
//  Long Run Test
//         Count runs of 34 or more in a row (FIPS 140-2 runs of 26 or more)
//         Fail if you get one.
//  RETURN CODES:
//    0 = passed
//    2 = failed self tests
//    3 = failed long runs test
//    4 = failed poker tests
//    5 = failed runs tests
//    6 = continuous RNG failed
private int SRNG_Test() //Perform SRNG Self Test
```

```
{
    int monobit_ones = 0; //the number of ones counted in the monobit test
    byte lastbit = 0x00; //the value of the last bit examined (one or zero)
    int runlength = 1; //the current size of the run of ones or zeroes examined
    int[] runs_ones={0,0,0,0,0,0,0}; //array counting the number of runs of ones
                 //runs_ones[6] contains all runs of 6 or greater
    int[] runs_zeros={0,0,0,0,0,0,0}; //array counting the number of runs of zeroes
    boolean longrun4 = false; //flag for encountering a run of 26 or more
    boolean pokerfail3 = false; //flag for failure of poker tests
    boolean runsfail2 = false; //flag for general failure of runs tests
    boolean testsfail1 = false; //flag for general failure of tests
    //counts the number of each four byte possibility for poker test
    int[] poker={0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0};
       byte[] random_sample={(byte)0x00}; //place to store a random byte for analysis
    byte shift_sample=0x00; //scratch byte for shifting values.
    int X = 0; //sum for poker test
    byte[] random_byte=new byte[500]; //storage for raw randoms

        for(int i=1; i<=5; i++) //loop for 5 500-byte samples from RNG
        {
        //get a 100 byte sample
        if (!getRandom(random_byte,25)) //generate random data (25*20 bytes)
        {
            return (6); //If continuous RNG test fails, fail entire test
        }

        for (int k=0;k<=499;k++) //loop for 500 bytes in sample
        {
            random_sample[0] = random_byte[k]; //examine a byte

            for (int j=7; j>=0; j--) //loop through each bit in our 8-bit byte
            {
                shift_sample = random_sample[0]; //make a copy for shifting
                shift_sample >>=j;  //look at the j'th bit of sample
                shift_sample &= 0x01; //mask off just that bit
                if (shift_sample==0x01) //if this bit is a one...
                    monobit_ones++; //add to the ones count
                if (shift_sample==lastbit) //if this bit extends a run
                    runlength++; //add to the runs count
                else  //ended a run, since new bit flipped from lastbit
                {
                    if (runlength >= 26) //if this is a "long run"
                        longrun4 = true; //flag the long run failure
                    if (runlength > 6) //lump runs higher than 6 into the 6+ category
                        runlength = 6; //set index into 6+ category
                    if (lastbit == 0x01)
                        runs_ones[runlength]++; //count the run in the appropriate index
                    else
                        runs_zeros[runlength]++; //count the run in the appropriate index
                    lastbit = shift_sample; //save the new run bit
                    runlength = 1;  //reset the runlength for the new bit.
                }
            //end loop on each bit of 8-bit sample
            }
            shift_sample = random_sample[0]; //make a copy of the sample
            shift_sample >>=4;  //rotate top half of sample to bottom
            shift_sample &=(byte)0x0F;  //look at only that part of sample
            poker[(int)shift_sample]++;  //add top half of byte as one poker sample
            shift_sample = random_sample[0]; //get a fresh copy of sample
            shift_sample &=(byte)0x0F;  //look at bottom half of sample
            poker[(int)shift_sample]++;  //add lower half of byte as one poker sample
        }//end loop on bytes in sample
    //end loop on 20,000 bit test
    }
    //check last run
    if (runlength >= 26) //if this is a "long run"
        longrun4 = true; //flag the long run failure
    if (runlength > 6) //lump runs higher than 6 into the 6+ category
        runlength = 6; //set index into 6+ category
    if (lastbit == 0)
        runs_zeros[runlength]++; //count the run in the appropriate index
    else
        runs_ones[runlength]++; //count the run in the appropriate index

    //check for failures
    if ((monobit_ones <= 9725) || (monobit_ones >= 10275) )
```

```
            testsfail1 = true; //throw error on monobit selftest fails
        for (int i = 0; i <=15; i++) //loop through poker test values
        {  //calculate X per FIPS 140-2 ( (sum of squares)*16/5000 - 5000 )
            X += poker[i]*poker[i];  //sum the squares
        }
        //We use 100 times formula so we can use int arithmetic and keep
        //two significant decimal places. (No Floats in iButton)
        X = X * 16 / 50 - 500000; //now we have 100 times the needed value
        //so we check against 100 times desired value
        if ((X <= 216) || (X >= 4617) )
            pokerfail3 = true; //throw error on poker selftest fails
        if ( //giant if statement on all the runs value bounds from FIPS 140-2
        (runs_zeros[1] <= 2315) || (runs_zeros[1] >= 2685) ||
        (runs_ones[1] <= 2315) || (runs_ones[1] >= 2685) ||
        (runs_zeros[2] <= 1114) || (runs_zeros[2] >= 1386) ||
        (runs_ones[2] <= 1114) || (runs_ones[2] >= 1386) ||
        (runs_zeros[3] <= 527) || (runs_zeros[3] >= 723) ||
        (runs_ones[3] <= 527) || (runs_ones[3] >= 723) ||
        (runs_zeros[4] <= 240) || (runs_zeros[4] >= 384) ||
        (runs_ones[4] <= 240) || (runs_ones[4] >= 384) ||
        (runs_zeros[5] <= 103) || (runs_zeros[5] >= 209) ||
        (runs_ones[5] <= 103) || (runs_ones[5] >= 209) ||
        (runs_zeros[6] <= 103) || (runs_zeros[6] >= 209) ||
        (runs_ones[6] <= 103) || (runs_ones[6] >= 209)
)
        //problem areas currently commented out.
        { //if any runs tests are out of bounds
            runsfail2 = true; //fail the runs tests
        }

        //return error codes in reverse order of likelihood
        if (testsfail1) // if general tests failed above
        {
            return (2); //return failure code
        }
        else if (longrun4)
        {
            return (3); //throw error on long runs selftest fails
        }
        else if (runsfail2)
        {
            return (5); //return error for runs tests
        }
        else if (pokerfail3)
        {
            return (4); //return error for poker tests
        }

        // all tests passed, return success
        return (0);
    }
    //END CONVENIENCE FUNCTIONS
} //End Class
```