Results and Code for a Software-based Implementation of ECDSA for Control System Data Authentication and Verification

Kenneth A. Fischer

Code 955, NSWCCD-SSES
1000 Kitty Hawk Avenue, Philadelphia PA 19112
Kenneth.a.fischer@navy.mil

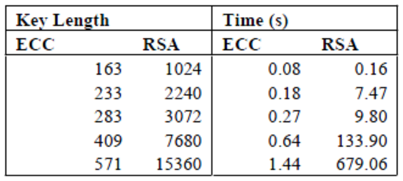
Abstract— *In previous works the need for control system data authentication and verification was discussed, and the use of the Elliptic Curve Digital Signature Algorithm (ECDSA) was proposed as a solution. Following up on that work, this paper presents a software based implementation of ECDSA utilizing ISaGRAF SoftPLCs running on both a virtual machine configuration (test string 1) and on four General Micro Systems VS 275 single board VME computers (test string 2). The implementation is primarily written in IEC 61131-3 ladder logic using specialized function blocks developed in C utilizing the OpenSSL library. The four SoftPLCs were configured in a round robin architecture in order to determine the average time it takes to generate a message, sign it, transmit it, receive it, and verify the signature. Test string 1 gave an average time of 26-28ms with PLC scan times of 22-24 ms. Test string 2 gave an average time of 55-60ms with PLC scan times of 55-60 ms. As expected, timing of the completed transmission is linearly related to PLC scan time.*

Index Terms—*PLC, HMI, Stuxnet, Smart Grid, NGIPS, ECDSA, Machinery Control System*

1. Introduction

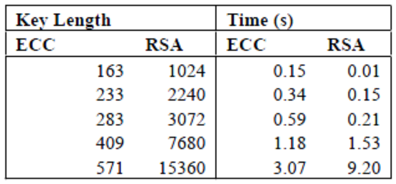
In previous works [1] the need for increased security for control systems was discussed. The requirements for control system security were surveyed and it was determined that a method for verifying the authenticity and integrity of the commands and data transmitted between PLCs was needed. Specifically, it was proposed that the data traveling across any controls network is first signed by the sending controller at the application layer. The signature would then be transmitted across the control network along with the data and be subsequently verified by the receiving controller at the application layer. This would allow the receiving controller to verify both the authenticity and integrity of the data and would allow both controllers to utilize any available communications protocol to transmit. It was also noted that signing and verifying the data at the application layer will also allow intermediary controllers to relay data silently without additional overhead that would impact transmission time.

To accomplish this, the use of a digital signature algorithm was proposed using a public key cryptography system. A literature survey was performed of two major cryptographic technologies used in digital signature algorithms: RSA and Elliptic Curve Cryptography (ECC). Figures 1 through 3 below show timing comparisons of the two different algorithms based on the work performed by Jansma and Arrendondo. The data showed that as higher levels of security are needed ECC technology begins to outperform RSA technologies. Given that the need for security will only increase, the Elliptic Curve Digital Signature Algorithm (ECDSA) was proposed as a solution for control system data authentication and verification.



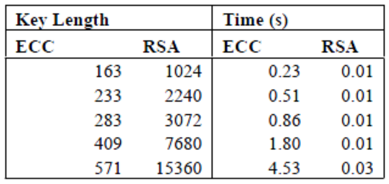
Key Length		Time (s)	
ECC	RSA	ECC	RSA
163	1024	0.08	0.16
233	2240	0.18	7.47
283	3072	0.27	9.80
409	7680	0.64	133.90
571	15360	1.44	679.06

Figure 1. ECC vs RSA Key Generation [2]



Key Length		Time (s)	
ECC	RSA	ECC	RSA
163	1024	0.15	0.01
233	2240	0.34	0.15
283	3072	0.59	0.21
409	7680	1.18	1.53
571	15360	3.07	9.20

Figure 2. ECC vs RSA Signature Generation [2]



Key Length		Time (s)	
ECC	RSA	ECC	RSA
163	1024	0.23	0.01
233	2240	0.51	0.01
283	3072	0.86	0.01
409	7680	1.80	0.01
571	15360	4.53	0.03

Figure 3. ECC vs RSA Signature Verification [2]

After reviewing the mathematics of ECDSA and potential variations of the algorithm, it was concluded that ECDSA is a promising solution and that an IEC61131-3 compliant implementation would be needed in order to use existing PLC hardware and encourage widespread adoption by control engineers. Unfortunately, such an implementation would be extremely complicated and expensive to produce. Furthermore, based on the timing data presented in [2], it seemed uneconomical to commit the resources towards development of such an implementation until the feasibility of the approach could be more fully evaluated and demonstrated.

SoftPLCs provide a unique opportunity to perform this analysis by developing a prototype system that is predominately written in IEC 61131-3 code but allows the use of specialized custom function blocks written in other high level languages. This allows the development of a prototype implementation of a control system ECDSA algorithm that is able to reuse existing software libraries in order to avoid the expense of developing ladder logic cryptographic functions. The prototype implementation, being predominately written in ladder logic, can then be reused for a full IEC611-31 implementation by simply replacing the custom function blocks with IEC61131-3 versions. Furthermore, the large number of SoftPLCs already existing in both Navy and industry applications would be better suited by the prototype implementation than the full IEC 61131-3 implementation.

This paper presents the source code, timing results, and stability results for an ISaGRAF SoftPLC implementation of ECDSA for control systems. ISaGRAF SoftPLCs are currently in use on a wide range of industrial control applications around the world, and are currently employed by the Navy in two major programs: the Littoral Combat Ship (LCS) Class and the Mobile Landing Platform (MLP) Class for ship wide machinery control. The challenges and solutions uncovered when developing the implementation are discussed, and a path forward for converting the prototype into a full IEC 61131-3 compliant implementation is presented.

2. SYSTEM ARCHITECTURE OVERVIEW

The primary goal of the prototype was to determine the overhead the ECDSA implementation would have on control system operation. In order to have effective control, the scan times for the logic must be sufficiently low enough to approximate real time operation (less than 100 ms and ideally less than 50 ms). Additionally, for PLC to PLC communications effective control requires that you are transmitting and processing at least one set of actions every 300 ms or less (ideally every 100 ms or less). There are a number of different things that can impact both scan times and transmission times, particularly for SoftPLCs running on a Windows OS, such as network latency and individual hardware I/O access rates.

In order to obtain averages for scan times and transmission times a simple “round robin” architecture was chosen as shown in Figure 4 below. Four SoftPLCs, labelled 1 through 4, were configured so that each processes a piece of data and then subsequently transmits it down the line, repeating the process in an indefinite loop. The data transmitted included a number of simulated signals, including BOOL, INT, REAL, DATE, and STRING values that were part of a small logic simulation routine. The details of this are discussed in Section 6 below.

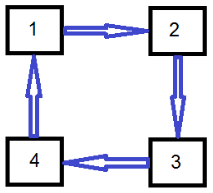


Figure 4. SoftPLC Round Robin Architecture Concept

The primary data point of interest for this system is a DINT value called COUNT. COUNT is a number that is initialized at 0 at the very start of the system. Each SoftPLC receives COUNT as an input from its predecessor, verifies the digital signature that came with the COUNT value, and if verification is successful it increments the COUNT value by 1, generates a new message, and transmits the new message to the next PLC in the loop. Figure 5 below illustrates the count increment lifecycle within the PLC’s execution processes. As can be seen, each value of count represents a complete set of signature generation, data transmission, and signature verification actions. Transmission time is then synonymous with the timing of the COUNT lifecycle. In order to determine an average time, the system was run until it had reached a certain value of COUNT. The runtime was then captured and the runtime divided by the COUNT gives us the average transmission time.

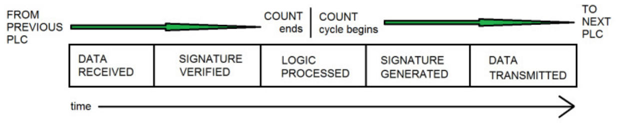


Figure 5. SoftPLC Round Robin Architecture Concept

Please note that some SoftPLC products offer an additional “WAIT” option as a step after the DATA TRANSMISSION step in their PLC. This feature normally works by predefining a PLC execution cycle time, like 100 ms, and essentially takes the 100 ms minus the time it took to complete all of the above steps and then waits for that time until it begins executing the next cycle. This is normally used to add a bit more determinism to the SoftPLC and provides a window of opportunity for the operating system to execute tasks. Since the goal of this prototype to determine loads by monitoring scan and transmission times the WAIT feature was disabled.

The PLC scan times would be monitored by using built-in tools that come with all major SoftPLC and regular PLC packages. Scan time is not synonymous with transmission time since the PLCs are not synchronized (as is typical with industrial operations). Additionally, it is possible that a PLC might receive an invalid signature in a data transmission due to events like network transmission errors that have damaged the data. Any COUNT with an invalid signature is disregarded, making COUNT representative for only successful transmissions with correct signature generation and verification.

3. SOFTWARE ARCHITECTURE BACKGROUND INFORMATION

3.1. SoftPLC Package

There are two popular SoftPLC packages currently in use by the Navy: Rockwell Canada's ISaGRAF package and Siemens WinAC. Both packages can be used to develop IEC 61131-3 ladder logic and include custom C function blocks. The primary difference between the packages is that Siemens WinAC includes additional functionality installed on the target hardware that adds real-time determinism and disables certain functions in Windows that could result in a "blue screen of death" error. ISaGRAF, on the other hand, is more comparable to Java in that it includes a simple executable known as the ISaVM that in turn executes the logic. Like Java, ISaGRAF has multiple ISaVM implementations that allow the software to be used with multiple platforms including both Linux and Windows. Ultimately ISaGRAF was chosen for this implementation because it used on more US Navy Ships than Siemens WinAC and because Rockwell Canada was willing to provide free licenses to support development of this prototype.

The ISaVM executable lives with a collection of related executables in a folder that is collectively referred to as the "target". The target is installed onto the hardware via a simple copy operation which can be performed anywhere within the file structure of the SoftPLC's operating system. The target includes a main program, "ISaGRAF.exe" that is manually started by the user. This program then starts up the subprograms including both ISaVM.exe and the default Ethernet communications program "ETCP.exe". The target also includes a text file known as the "target definition" file that describes details about the target such as what custom C function blocks are available.

In order to develop custom C function blocks for inclusion in the ladder logic a tool called TDBuild (Target Definition Builder) is used to define the input and output variables of the function blocks. The tool is then used to auto-generate C code which essentially provides the API between the ladder logic and the custom C code that was developed as part of this prototype. TDBuild is also used to update the target definition files, which are then read into the main ISaGRAF program in order to allow the software to include the specialized function blocks in the ladder logic.

For Windows-based targets the custom C code is eventually compiled as a dynamic link library (DLL) which is then copied and pasted into the target folder. This DLL, combined with the target definition files discussed previously, provide full runtime access to the custom C code for program execution. Note that you do not have to register the DLL with Windows which greatly simplifies the install.

3.2. OpenSSL

As stated previously, developing a custom IEC 61131-3 implementation of ECDSA is a complicated endeavour and a prototype system that utilizes existing implementations is a necessary first step. For this prototype the OpenSSL [3] implementation has been chosen for both its efficiency and its free and open source availability to all controls engineers. There are three methods by which OpenSSL can be included in a project:

1. By installing OpenSSL onto the target system and using its command line interface.
2. By performing a fresh compile of OpenSSL as a dynamic library which can then be copied onto the target system and used after first properly registering the DLL.
3. By performing a fresh compile of OpenSSL as a static library which can then be included in your application without having to first install OpenSSL onto the target system or register any DLLs including the DLL used by the ISaGRAF software itself.

For this prototype OpenSSL was compiled as a static library in order to simplify the installation process onto the target machines (i.e. to keep the ISaGRAF target install down to a simple copy operation). The downside to this option is that it does complicate the structure of the actual C source code and increased development time of the prototype. It was decided that the increase in development time is justified by the increased ability to use the prototype on existing Navy ship classes that are using SoftPLCs and do not require a 100% IEC 61131-3 implementation, such as LCS and MLP.

The following steps were performed to compile OpenSSL as a static library on a Windows 7 machine:

1. Install Visual Studio 2010
2. Install Active Pearl 32-bit
3. Download the latest version of OpenSSL
4. Open the Visual Studio Command Prompt
5. Unzip OpenSSL to a directory such as C:\openssl-src-32
6. CD to the directory

7. Run the command: perl Configure VC-WIN32 –prefix=C:\Build-OpenSSL-VC-32
8. Run the command: ms\do_ms
9. Run the command: nmake –f ms\nt.mak
10. Run the command: nmake –f ms\nt.mak install

The compiled library will end up in the C:\Build-OpenSSL-VC-32 directory.

3.3. Visual Studio 2010 Express

Visual Studio 2010 Express (a free product) was used to develop and compile the custom C function blocks into a DLL. Taking the C code and header files generated by TDBuild and adding it to the Visual Studio project is a simple matter, but there are additional steps that must be taken in order to configure the compiler to properly utilize the dependent libraries in order to generate the DLL. The two critical steps required are:

1. Under Linker – General – Additional Library Dependencies add the OpenSSL static library.
2. Under Linker – Input – Additional Dependencies add the libeay32.lib and the ssleay32.lib.

3.4. Cryptographic Algorithms Used

As discussed in the previous work [1] performing a digital signature requires the use of two different cryptographic algorithms: a hash function and a public-key cryptography function. The hash function is performed first in order to process the arbitrarily long amount of data into a fixed length tag. The cryptographic function then encrypts the data using the private key, which can then be decrypted using the public key by an agent desiring to verify the public signature.

For this work the SHA-512 algorithm was used in combination with the P-521 ECDSA algorithm. The goal was to use the strongest measure of security possible at the time of development, and it is worth noting that the weaker P-384 ECDSA algorithm is considered valid for even TOP SECRET data. OpenSSL includes implementations of both algorithms in the static library that was compiled.

Using the SHA-512 algorithm was relatively trivial, but the P-521 algorithm proved to be a bit tricker than originally anticipated. The P-521 algorithm consists of 65 bytes of data plus 1 bit. The extra bit, when true, results in an output of 66 bytes. When the extra bit is false, the result is an output of 65 bytes. As will be illustrated in the walkthrough of the C code (and as is shown in the source code comments) special considerations were needed to monitor the output of the algorithm to ensure that the correct amount of data was read.

3.5. Base64 Encoding / Decoding

Transmitting cryptographic keys and digital signatures between PLCs is not a trivial manner. PLCs only include a predefined number of data types such as BOOL, INT, DINT, and STRING which can be transmitted between devices. All of the data types except for STRING are therefore unsuitable for transmitting the keys and signatures due to the insufficient bit length.

In most PLC products, the STRING variable can consist of any ASCII character array up to a length of 255 characters according to the literature of the various vendors. This would appear to give a data value of 2040 bits using ASCII encoding, sufficiently long enough to transmit ECDSA keys and signatures. Note that this length is still too short for the use of RSA keys and digital signatures which is one of the reasons why ECC technologies were chosen.

The problem is that the 255 character set is misleading, since in ISaGRAF and in other PLC products 3 of the 255 characters are reserved to process the STRING. One of the three character slots must be used for the NULL character (which should indicate to the experienced C programmer that the STRING data type is really a C-String and not the more advanced string data type found in C++ or other high level languages). Additionally, the STRING must begin with an apostrophe and end with an apostrophe. The "apostrophe" problem becomes our greatest concern. When representing an ECDSA signature or key in ASCII it is possible that the resulting ASCII string will output an apostrophe in the middle of the key or signature. The PLC will interpret the second apostrophe (and note that you must always start with an apostrophe) as the end of the string and subsequently cut off the remaining data.

In order to work around this problem, a different character encoding must be used that maximizes the data compression of the string representation. Hexadecimal representation of the data was considered, but at only 4 bits per character it was considered insufficient. A custom variant of ASCII was considered that didn't use the apostrophe, but in the interests of conforming to widely accepted standards this idea was dropped. Base64 encoding, using 6 bits per character, became the most logical choice.

Base64 encoding allows the cryptographic keys to be transmitted as a single string variable for each key. The private key is represented by a 75 character string (plus 3 for the null and the leading/lagging apostrophes). The public key, which represents a point on an elliptic curve, is represented by a 152 character string (plus 3). The digital signature is represented by two values, SIG_R and SIG_S, each with a length equivalent to the private key. These values were transmitted as two separate strings to help facilitate the need for possible future expansion.

Note that the STRING variables themselves which are storing the data in the PLC ladder logic are required to be set to a fixed length. This is a fairly standard requirement by most PLC manufacturers. In order to facilitate the need for future strong cryptography it was decided to set the length for these variables in the ladder logic to the largest possible value. As a consequence, the PLC STRING variables include padded data in addition to the cryptographic keys or digital signatures.

3.6. Self-Signing Keys

One of the largest areas of discussion with using a public-key cryptography system for control system security has been a concern about how to setup a public-key infrastructure (PKI). In order for two systems to be able to communicate securely, there needs to be some initial trust relationship established so that a receiving system which obtains a public key from a sending system knows that the public key really belongs to that system. Put another way, how does the receiver really know that the sender is who it claims to be?

In traditional information systems a Certificate Authority (CA) is used to issue certificates to users. This certificate contains the identity of the key pair owner, the owners public key, and a digital signature of the Certificate Authority. When users communicate securely, they receive a copy of each other's signed certificate (minus the private key) and they establish trust in each other's identity based on the successful validation of the CA's signature. Put another way, they trust each other because someone else told them it was okay. Ultimately, even trusting the digital signature of a CA is based on the idea that some human ultimately made the decision that the CA who issued the certificates is of good repute.

This kind of complicated infrastructure is necessary in IT systems, since two communicating systems may send a wide range of different kinds of data with a wide disparity in both timing and content. Furthermore, an IT system may communicate only once with another system it never heard of before and then never communicate to that system again. This constant flux requires a PKI in order to establish trust relationships with CAs and individual users. Unfortunately, running a PKI system requires a significant amount of processing power and adds significant complexity to a control system. Significant research is ongoing to develop a modified version of a PKI that can be executed on operational technology platforms, however most of these are simply slightly modified versions of the same technologies used in IT systems.

It is the opinion of the author that implementing a PKI for control systems is unnecessary. The two central problems that a PKI system solves are 1) establishing a trust relationship between two systems and 2) handling the large amount of flux in data transmitted between control systems. In Operational Technology systems, neither of these problems really exists. OT systems are designed to communicate in a very consistent and precise manner, transmitting the same basic message structure at a consistent interval to the exact same targets. The relationship between these targets is established when the system is first commissioned for operation (i.e. on the day of birth) and no new communication partnerships are ever established without significant software changes and a recommissioning of the system.

Therefore, in designing this prototype, a different system for establishing communications and updating keys was created. During commissioning, each system is preconfigured with a public / private key pair, and the public keys of each part of the system are given to each of the other parts. When an agent in the system decides to change its public / private key pair (this decision is made at regular intervals with the interval length configured at commissioning) it first generates a new key pair and then signs the new public key with the old private key. This information is then transmitted to each of the other agents in the system who then verify the key change message with the old public key they currently have on file. If the digital signature is valid, the new public key is accepted and the old public key is disregarded.

In order to ensure that an adversary will not be able to attack the system using the original preconfigured public/private key pair each agent immediately changes its key to a new random key pair at startup, before any other logic is processed. Additionally, since there is no CA and communications are highly deterministic there is little risk in changing the key pairs at a much more frequent interval. In the prototype, each of the four SoftPLC agents were configured to change their key pairs at 53 minutes, 59 minutes, 61 minutes, and 67 minutes respectively.

As a result, this system effectively generates keys 175,000 to 265,000 times more often the PKI CA systems used in IT applications and greatly weakens an adversaries capabilities to brute force crack the system. Ultimately, this allows for smaller key sizes to be used by OT systems and thus reduces the required processing power and memory required to effectively implement an ECDSA algorithm.

The major downside with this implementation is that an agent in the control system which is powered down must retain their current public/private key pair in memory and resume use of that key pair upon startup (though it can immediately change it). A agent which does not have this capability may default back to the original pre-commissioned key pair and will therefore be considered to be a bad actor by the other agents in the system, requiring a complete re-initialization of the entire system. Fortunately, this problem is easily solved by adding the required memory capabilities and is mitigated by the fact that these kinds of systems are highly redundant and designed to run uninterrupted for years without failures.

4. HARDWARE ARCHITECTURE

Two test strings were used in the development and testing of the prototype, Test String 1 and Test String 2.

4.1. Test String 1

Test String 1 used four Windows 7 virtual machines running on VMWare Player on top of a Windows 7 Pro 64-bit machine. This system was primarily used for initial development and testing. The ISaGRAF software was run on the main machine in order to view the PLC code. The four virtual machines were bridged to the host NIC card which was then connected to a router. DHCP was used to assign IP addresses to both the host and virtual machines.

The main machine had the following relevant hardware specifications:

- I5-2500K processor, 4C, overclocked to 4.3 GHz, with 4x256KB of L2 Cache and 6MB of L3 Cache
- 16 GB RAM
- 1 GB Hardwired NIC
- 1.5 TB SATA 3 HDD

4.2. Test String 2

Test String 2 used four General Micro System VS275 Single Board Computer VME boards in the same VME chassis. No communications across the VME backplane were used. Each board was running Windows XP Professional, 32-bit, SP2. Figure X below shows the front panel of the VS275.

Figure 6. SoftPLC Round Robin Architecture Concept

Each VS275 board had the following specifications:

- 2.16 GHz Core 2 Duo Processor, 4-MB L2 cache
- 3 GB of 667-MHz DDR-2 SDRAM
- 1 GB Hardwired NIC
- 64 GB SATA2 SSD

Hardwired Ethernet communications were used via the ENET port on the front of the board. All four boards were connected to an 8 port flat 10/100 MB hub that was in turn connected to a 10/100/1000MB router. A development station running ISaGRAF was connected to the router directly in order to download the software to the boards and to go online to the SoftPLCs. DHCP was used to assign IP addresses to each of the boards and development station.

5. SOURCE CODE WALKTHROUGH

Copies of the source code developed for this prototype with comments are available as appendices to this paper, and key details and decisions made in designing the software have been presented in the previous sections. This subsection presents a high level description of each of the modules that together make up the prototype, in an effort to provide context for the source code modules.

Note that for the C code, the portions of the ISaGRAF code generated by the TDBuild tool are not available as that code is considered proprietary. Fortunately, the code not shown simply serves as the API to the PLC ladder logic and is specific to the ISaGRAF platform. Current users of ISaGRAF will be able to use the same tools with the information provide in this document to quickly regenerate this code. Users of other SoftPLC products should be able to develop their own variations using the information provided in this paper.

5.1. C CODE Walkthrough

The subsections below describe the model and C code for each of the custom function blocks that are used in the IEC61131-3 ladder logic. A fully compliant IEC61131-3 implementation will replace the C code of these blocks with a ladder logic version.

In developing the prototype, debug logic was added into each of the function blocks. Two different mechanisms for debug logic were included. The first is that each block outputs an integer status variable, with a 1 indicating successful operation and a negative number indicating a failure. The code of the blocks is organized internally in steps as shown in the source code and comments. A failed step will output a negative number equal to the step number (i.e. a fail on step 3 will output a -3 value for the status).

The second debug logic is normally turned off for full operation and was only added for development. This logic generates text files during operation that contain key status variables useful for debugging the code. This adds overhead to the program execution and creates a security risk, but is extremely valuable for debugging. In order to turn the logic on or off, the variable "ISaDEBUG" must be set true for on or false for off. This variable is located in the "debug.h" file, and a change in the status of the variable requires a recompile of the DLL. All the timing results presented in this paper are with the ISaDEBUG variable set to false.

5.1.1. MsgGen

Figures 7 and 8 below define the details of the MsgGen block. In digital signature applications, the first step of an algorithm is to hash the data to be signed into one single value of fixed length, that will later be encrypted by the private key. The MsgGen block generates that hash value using the SHA-512 algorithm. The data to be signed is inputted into the B#, I#, R#, and STRING fields. The DATE and TIME fields should always be used for the current date and time stamps of the message or the entire algorithm is potentially subject to a replay attack. The block outputs the hash as a hexadecimal string.

Creating one generic message generation block that can be used in a wide range of applications is tricky, since one application could only need a few Booleans and another application could require 40 different real values. Therefore the block was structured generically in order to allow the widest range of possible inputs, and then a HASH_I value was added that allows the user to chain multiple blocks together. Readers familiar with the SHA-512 algorithm will recognize that there is no loss in processing the data in this manner, since the algorithm generates the hash in a "chain" fashion naturally. This implementation simply extends the chain, and there is no limit to the number of MsgGen blocks that can be chained together.

The key for implementing this function is that the data to be signed must have a hash generated on both the source PLC and the destination PLC. Therefore, during system design, it is critical that the layout of the MsgGen blocks is the same on each endpoint or the signature will never verify successfully.

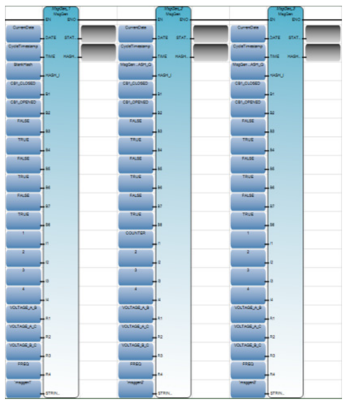


Figure 7. Three MsgGen Blocks Connected Together

Variable Name	I/O	Type	Description
DATE	IN	DATE	Used to date the message in order to prevent replay attacks. Has resolution up to the second.
TIME	IN	TIME	Used to timestamp the message in order to prevent replay attacks occurring within the last second. Has resolution up to the millisecond.
HASH_I	IN	STRING(252)	Hash from previous MsgGen. Used to string multiple blocks together in order to generate messages containing large amounts of data.
B1 Thru B8	IN	BOOL	Boolean data inputs.
I1 Thru I4	IN	DINT	Integer data inputs.
R1 Thru R4	IN	REAL	Real data inputs.
STRING	IN	STRING(252)	String data inputs.
STATUS	OUT	INT	Status of the computation.
HASH_Q	OUT	STRING(252)	Hash of the message.

Figure 8. MsgGen Input / Output Structure

5.1.2. KeyGen

The KeyGen block generates an ECC-521 prime public/private key pair. Ladder logic should be placed in the application to only enable the block at specific intervals when generation of a new key is desired. The new keys are outputted as base64 encoded strings. Figures 9 and 10 below define the details of the KeyGen block.

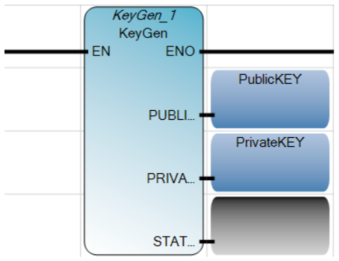


Figure 9. KeyGen Block

Variable Name	I/O	Type	Description
PUBLIC_KEY	OUT	STRING(252)	ECC 521 public key
PRIVATE_KEY	OUT	STRING(252)	ECC 521 private key
STATUS	OUT	INT	Status of the computation

Figure 10. KeyGen Input / Output Structure

5.1.3. KeyVerify

The KeyVerify block is used as an added check to verify the integrity of the public/private key pair by using the private key to first sign an internal dummy hash and then using the public key to verify the signature. This block is solely used in the prototype to verify that the system is functioning correctly. As the maturity of the design improves it is believed that this block can be removed, thus reducing system overhead and improving communications results. Figures 11 and 12 below define the details of the KeyVerify block.

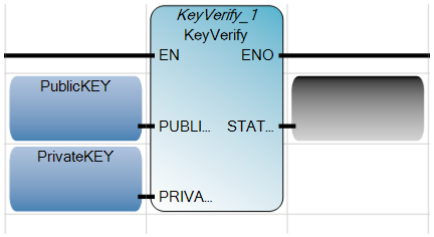


Figure 11. KeyVerify Block

Variable Name	I/O	Type	Description
PUBLIC_KEY	IN	STRING(252)	ECC 521 public key
PRIVATE_KEY	IN	STRING(252)	ECC 521 private key
STATUS	OUT	INT	Status of the computation

Figure 12. KeyVerify Input / Output Structure

5.1.4. SigGen

The SigGen block is the heart of the ECDSA implementation, and takes the hash from the MsgGen block and the PrivateKey from the KeyGen block to output an ECDSA P-521 signature. The signature itself is composed of two components, SIG_R and SIG_S, which are each a base64 encoded string. Figures 13 and 14 below define the details of the SigGen block.

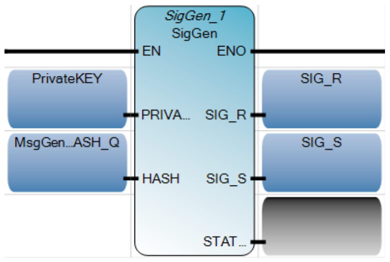


Figure 13. SigGen Block

Variable Name	I/O	Type	Description
PRIVATE_KEY	IN	STRING(252)	ECC 521 private key
HASH	IN	STRING(252)	Data hash to be signed from the MsgGen block.
SIG_R	OUT	STRING(252)	ECDSA P-521 signature, R component.
SIG_S	OUT	STRING(252)	ECDSA P-521 signature, S component.
STATUS	OUT	INT	Status of the computation

Figure 14. SigGen Input / Output Structure

5.1.5. SigVerify

The SigVerify block is used to verify that the validity of a digital signature by the receiving agent. The receiving agent first uses the MsgGen block to generate the same hash that the original sender generated, and then uses the public key of the sender along with the signature to determine if the signature and the hash match. Figures 15 and 16 below define the details of the SigGen block.

The SigVerify block is also used in the prototype system on the sender side to ensure a signature is properly generated before transmission. As the design matures it is believed that this usage of the block can be eliminated, reducing system overhead and improving communications results.

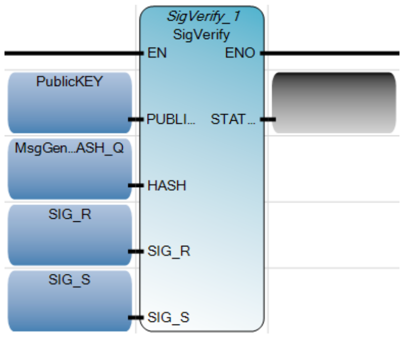


Figure 15. SigVerify Block

Variable Name	I/O	Type	Description
PUBLIC_KEY	IN	STRING(252)	ECC 521 public key
HASH	IN	STRING(252)	Data hash to be verified from the MsgGen block.
SIG_R	IN	STRING(252)	ECDSA P-521 signature, R component.
SIG_S	IN	STRING(252)	ECDSA P-521 signature, S component.
STATUS	OUT	INT	Status of the computation

Figure 16. SigGen Input / Output Structure

5.2. LADDER LOGIC WALKTHROUGH

For the prototype system, the ladder logic for each of the four SoftPLC agents is structured identically, and execution of the logic commences in the following order:

1. ECDSA_D#_I – process digital signatures for incoming data transmissions, including the validation of new public keys sent to the SoftPLC
2. MAIN – general logic such as determining current date, setting ALWAYS_ON and ALWAYS_OFF bits, and incrementing the counter value received during ECDSA_D#_I
3. SIM_BOOL and SIM_REAL – generic simulation routines that fill the place for where actual PLC logic would normally occur
4. ECDSA_KeyCntrl – controls the generation and validation of the SoftPLC's own private / public key pair
5. ECDSA_Q – generates digital signatures for the outgoing data transmissions

6. RESULTS AND FUTURE WORK

The prototype implementation was run on both test strings for periods ranging from several days to several weeks. COUNT values, reflecting the number of completed message transmissions (including both the signature generation and signature verification components) ranged from 10,000 to 4 million. In all scenarios the timing results were independent of the count length for values over 10,000. Test String 1 gave an average time of 26-28ms with PLC scan times of 22-24 ms. Test string 2 gave an average time of 55-60ms with PLC scan times of 55-60 ms. As expected, timing of the completed transmission is linearly related to PLC scan time.

Values under 10,000 counts gave slightly higher results (1-5 ms increase). During the first scan of the PLCs when the system is initializing there is an additional load on the system to establish TCP communications resulting in an increased scan time and transmission time for that cycle. For low COUNT values this initial time has a stronger effect on the average. As a result, for approximately the first 10 minutes of system runtime the average transmission time gradually decreased until the system settled at the values stated above.

7. CONCLUSION

It is believed that the timing results are the worst case scenario given that no real-time modifications were added to the operating system (which would be untypical for a SoftPLC control system). Since the entire timing cycle is in the order of tens of milliseconds (as opposed to seconds as published by [2]) future development of a fully IEC 61131-3 compliant prototype is justified.

REFERENCES

[1] Fischer, K., "Control System Data Authentication and Verification Using Elliptic Curve Digital Signature Algorithm." Presented at ASNE Intelligent Ships Symposium X, May 22-23, 2013, Philadelphia, PA.

[2] Jansma, N., Arrendondo, B. (2004). "Performance Comparison of Elliptic Curve and RSA Digital Signatures." (http://nicj.net/files/performance_comparison_of_elliptic_curve_and_rsa_digital_signatures.pdf) Accessed: 12th July 2012

[3] OpenSSL. http://www.openssl.org/ Accessed: 24 February 2014

ACKNOWLEDGEMENTS

The author would like to thank Dr. Richard Perry, Villanova Unversity, Associate Professor of Electrical and Computer Engineering, for his support, technical review, and mentorship in this work.

The author would like to thank Stefan Mizera, Rockwell Automation Canada – ISaGRAF, for his company's support in providing free ISaGRAF licenses and technical support for this work.

Kenneth A. Fischer, received a BS in Chemical Engineering from the University of Delaware and is pursuing an MS in Computer Engineering from Villanova University. Mr. Fischer has over 10 years of automation and controls experience in pharmaceutical, power generation, food and beverage, specialty chemical, and naval applications. He is currently employed with NSWCCD-SSES Code 955 providing engineering support to the INLS, ENFMC, LCS, and DDG1000 programs.

APPENDIX A – C SOURCE CODE SAMPLE

*NOTE: Only the parts of the C Source Code written by the author are included in this Appendix. Auto-generated code from TD Build will not be included. Also note that header files not included indicate that the file is completely the result of auto-generated code. Presence of auto-generated code in the source files below will be indicated by the tag *---autocode---*.

A.1 - MsgGen

```
/***************************************************************************
File:                   msggen.c
Author:                 kfischer
Creation date:          05/11/2013 - 19:52
POU name:               MsgGen
***************************************************************************/

---autocode---

//START OF CUSTOM CODE HEADERS
#include <debug.h>
#include <stdio.h>
#include <stdlib.h>
#include <openssl/sha.h>          //for SHA512
//END OF CUSTOM CODE HEADERS

---autocode---


   //START OF CUSTOM CODE

   //--Declarations
    /*debug variables*/
       #if ISaDEBUG
              FILE *fp;
       #endif

       /*string and hash variables*/
    #define P_PTR_HASH_I_DATA (((char*)VA_ADDRESS(pvBfData,_P_IEC_DATA->hash_i))+2)
       #define P_PTR_STRING_I_DATA (((char*)VA_ADDRESS(pvBfData,_P_IEC_DATA-
>string))+2)
       unsigned char digest[SHA512_DIGEST_LENGTH];
       char mdString[SHA512_DIGEST_LENGTH*2+1];
       char *hash_Q;
       const char *string;
       char *tstring;
       char str[STRLEN];
       char tstr[STRLEN/4];
       int i;

       /*base64 variables*/

   //--Logic

       /*STEP 0 - Initialization*/
       P_STATUS = 0;
```

```
memset(digest,0,SHA512_DIGEST_LENGTH);
memset(mdString,0,SHA512_DIGEST_LENGTH*2+1);
memset(str,0,STRLEN);
memset(tstr,0,STRLEN/4);
hash_Q = mdString;
string = str;
tstring = tstr;

/*STEP 1 - build string for hashing*/
strcpy(str, "DATE=");
sprintf(tstring,"%u,",P_DATE);
strcat(str,tstring);

strcat(str, "TIME=");
sprintf(tstring,"%u,",P_TIME);
strcat(str,tstring);

strcat(str, "HASH_I=");
sprintf(tstring,"%s,",P_PTR_HASH_I_DATA);
strcat(str,tstring);

strcat(str,"P_B1=");
sprintf(tstring,"%s,", (P_B1)?"TRUE":"FALSE");
strcat(str,tstring);

strcat(str,"P_B2=");
sprintf(tstring,"%s,", (P_B2)?"TRUE":"FALSE");
strcat(str,tstring);

strcat(str,"P_B3=");
sprintf(tstring,"%s,", (P_B3)?"TRUE":"FALSE");
strcat(str,tstring);

strcat(str,"P_B4=");
sprintf(tstring,"%s,", (P_B4)?"TRUE":"FALSE");
strcat(str,tstring);

strcat(str,"P_B5=");
sprintf(tstring,"%s,", (P_B5)?"TRUE":"FALSE");
strcat(str,tstring);

strcat(str,"P_B6=");
sprintf(tstring,"%s,", (P_B6)?"TRUE":"FALSE");
strcat(str,tstring);

strcat(str,"P_B7=");
sprintf(tstring,"%s,", (P_B7)?"TRUE":"FALSE");
strcat(str,tstring);

strcat(str,"P_B8=");
sprintf(tstring,"%s,", (P_B8)?"TRUE":"FALSE");
strcat(str,tstring);

strcat(str,"P_I1=");
sprintf(tstring,"%d,",P_I1);
strcat(str,tstring);

strcat(str,"P_I2=");
sprintf(tstring,"%d,",P_I2);
```

```
strcat(str,tstring);

strcat(str,"P_I3=");
sprintf(tstring,"%d,",P_I3);
strcat(str,tstring);

strcat(str,"P_I4=");
sprintf(tstring,"%d,",P_I4);
strcat(str,tstring);

strcat(str,"P_R1=");
sprintf(tstring,"%f,",P_R1);
strcat(str,tstring);

strcat(str,"P_R2=");
sprintf(tstring,"%f,",P_R2);
strcat(str,tstring);

strcat(str,"P_R3=");
sprintf(tstring,"%f,",P_R3);
strcat(str,tstring);

strcat(str,"P_R4=");
sprintf(tstring,"%f,",P_R4);
strcat(str,tstring);

strcat(str, "P_STRING = ");
sprintf(tstring,"%s,",P_PTR_STRING_I_DATA);
strcat(str,tstring);

if (NULL == str)
{
       P_STATUS = -1; /*failed to create string*/
}
else
{
       /*STEP 2 - hash string*/
       SHA512_CTX ctx;
       SHA512_Init(&ctx);
       SHA512_Update(&ctx, string, strlen(string));
       SHA512_Final(digest, &ctx);

       for(i = 0; i < SHA512_DIGEST_LENGTH; i++)
       {
              sprintf(&mdString[i*2], "%02x", (unsigned int)digest[i]);
       }

       if (NULL == mdString)
       {
              P_STATUS = -2; /*failed to create hash*/
       }
       else
       {
              /*STEP 3 - output result*/
              strcpy(P_HASH_Q,hash_Q);
              HASH_Q_MAXLEN = 128;
              HASH_Q_CURLEN = 128;
              P_STATUS = 1;
       }
```

```
        }



        /*debug logic*/
        #if ISaDEBUG
           fp = fopen("debug-message.txt", "w");
           if (fp == NULL) {
                  P_STATUS = 666;
                  exit(0);
           }

           fprintf(fp, "string = %s\n", string);
           fprintf(fp, "digest = %s\n", digest);
           fprintf(fp, "mdString = %s\n", mdString);


           fprintf(fp, "\n\nDATE = %u\n", P_DATE);
           fprintf(fp, "HASH_I = %s\n", P_PTR_HASH_I_DATA);
           fprintf(fp, "BOOL1 = %s\n", (P_B1)?"TRUE":"FALSE");
           fprintf(fp, "BOOL2 = %s\n", (P_B2)?"TRUE":"FALSE");
           fprintf(fp, "BOOL3 = %s\n", (P_B3)?"TRUE":"FALSE");
           fprintf(fp, "BOOL4 = %s\n", (P_B4)?"TRUE":"FALSE");
           fprintf(fp, "BOOL5 = %s\n", (P_B5)?"TRUE":"FALSE");
           fprintf(fp, "BOOL6 = %s\n", (P_B6)?"TRUE":"FALSE");
           fprintf(fp, "BOOL7 = %s\n", (P_B7)?"TRUE":"FALSE");
           fprintf(fp, "BOOL8 = %s\n", (P_B8)?"TRUE":"FALSE");
           fprintf(fp, "INT1  = %i\n", P_I1);
           fprintf(fp, "INT2  = %i\n", P_I2);
           fprintf(fp, "INT3  = %i\n", P_I3);
           fprintf(fp, "INT4  = %i\n", P_I4);
           fprintf(fp, "REAL1 = %f\n", P_R1);
           fprintf(fp, "REAL2 = %f\n", P_R2);
           fprintf(fp, "REAL3 = %f\n", P_R3);
           fprintf(fp, "REAL4 = %f\n", P_R4);

           fclose(fp);
        #endif


    //END OF CUSTOM CODE


}

/* eof *********************************************************************/
```

A.2 – KeyGen

```
/***************************************************************************
File:               keygen.c
Author:             kfischer
Creation date:      15/06/2013 - 20:02
POU name:           KeyGen
***************************************************************************/

---autocode---

//START OF CUSTOM CODE HEADERS
#include <debug.h>
#include <stdio.h>
#include <stdlib.h>
#include <openssl/ec.h>      // for EC_GROUP_new_by_curve_name, EC_GROUP_free,
EC_KEY_new, EC_KEY_set_group, EC_KEY_generate_key, EC_KEY_free
#include <openssl/obj_mac.h> // for NID_secp384r1
#include <openssl/bn.h>             // for BIGNUM conversions
#include <openssl/bio.h>
#include <openssl/evp.h>     // for base64 conversions
#include <openssl/buffer.h>
#include <math.h>
#include <base64.h>                 // for base64 conversions

//END OF CUSTOM CODE HEADERS
---autocode---

   //START OF CUSTOM CODE

   //--Declarations


    /*debug variables*/
      #if ISaDEBUG
             FILE *fp;
             FILE *fpa;
             FILE *fpb;
             int i,j;
      #endif

   /*status variables*/
   const int set_group_success = 1;
   const int gen_success = 1;
   int set_group_status;
   int gen_status;

   /*key and base 64 variables*/

   const BIGNUM *PrivateKeyBN;
   BIGNUM *PublicKeyBN;
   const EC_POINT *PublicKeyPoint;
   EC_KEY *eckey;
   EC_GROUP *ecgroup;

   unsigned char *PublicKeyBinaryUnsigned, PublicKeyBinaryUnsignedArray[250];
   unsigned char *PrivateKeyBinaryUnsigned, PrivateKeyBinaryUnsignedArray[250];

   int encodedSize;
```

```
    char *buffer, BufferArray[250];
    unsigned char *PrivateKeyBase64, PrivateKeyBase64Array[250];

    int encodedSize_b;
    char *buffer_b, BufferArray_b[250];
    unsigned char *PublicKeyBase64, PublicKeyBase64Array[250];


    //--Logic

    /*STEP 0 - Initialization*/
        P_STATUS = 0;

        memset(PrivateKeyBinaryUnsignedArray,0,250);
        PrivateKeyBinaryUnsigned = PrivateKeyBinaryUnsignedArray;

        memset(PublicKeyBinaryUnsignedArray,0,250);
        PublicKeyBinaryUnsigned = PublicKeyBinaryUnsignedArray;

        memset(PrivateKeyBase64Array,0,250);
        PrivateKeyBase64 = PrivateKeyBase64Array;

        memset(PublicKeyBase64Array,0,250);
        PublicKeyBase64 = PublicKeyBase64Array;

        memset(BufferArray,0,250);
        buffer = BufferArray;
        memset(BufferArray_b,0,250);
        buffer_b = BufferArray_b;

        /*STEP 1 - create key object*/
     eckey=EC_KEY_new();
     if (NULL == eckey)
     {
         P_STATUS = -1;  /*failed to create key object*/
     }
     else
     {
               /*STEP 2 - create EC_GROUP object*/
         //ecgroup = EC_GROUP_new_by_curve_name(NID_secp384r1);    --worked for 384
  ECDSA, will want to come back and add an option to turn this on and off
               ecgroup = EC_GROUP_new_by_curve_name(NID_secp521r1);
         if (NULL == ecgroup)
         {
             P_STATUS = -2; /*failed to create new EC Group*/
         }
         else
         {
                      /*STEP 3 - associate key with group*/
             set_group_status = EC_KEY_set_group(eckey,ecgroup);
             if (set_group_success != set_group_status)
             {
                             P_STATUS = -3; /*failed to associate key with group*/
             }
             else
                      {
                             /*STEP 4 - create PRIVATE and PUBLIC keys*/
                 gen_status = EC_KEY_generate_key(eckey);
                 if (gen_success != gen_status)
```

```
                {
                    P_STATUS = -4; /*failed to generate EC Key*/
                }
                else
                            {
                                    /*STEP 5 - extract PRIVATE keys*/
                                    PrivateKeyBN = EC_KEY_get0_private_key(eckey);
                                    if (PrivateKeyBN == NULL)
                                    {
                                           P_STATUS = -5; /*failed to extract PRIVATE
key*/
                                    }
                                    else
                                    {
                                           /*STEP 6 - extract PUBLIC KEY*/
                                           PublicKeyPoint =
EC_KEY_get0_public_key(eckey);
                                           if (PublicKeyPoint == NULL)
                                           {
                                                  P_STATUS = -6; /*failed to extract
PUBLIC key*/
                                           }
                                           else
                                           {
                                                  /*STEP 7a - Private Key convert from
BN to binary and then encode as base64*/


       BN_bn2bin(PrivateKeyBN,PrivateKeyBinaryUnsigned);
                                                  encodedSize =
EVP_EncodeBlock(PrivateKeyBase64,PrivateKeyBinaryUnsigned,89);

                                                  /*STEP 7b - Public Key convert from
Point to BN to binary and then encode as base64*/

                                                  PublicKeyBN = BN_new();

       EC_POINT_point2bn(ecgroup,PublicKeyPoint,POINT_CONVERSION_UNCOMPRESSED,PublicKe
yBN,NULL);

       BN_bn2bin(PublicKeyBN,PublicKeyBinaryUnsigned);
                                                  encodedSize_b =
EVP_EncodeBlock(PublicKeyBase64,PublicKeyBinaryUnsigned,179);



                                                  /*STEP 7c - set outputs*/

                                                  buffer = (char*) PrivateKeyBase64;
                                                  buffer_b = (char*) PublicKeyBase64;


                                                  strcpy(P_PRIVATE_KEY_BUFADD, buffer);
                                                  PRIVATE_KEY_MAXLEN = 250;
                                                  PRIVATE_KEY_CURLEN = encodedSize;

                                                  strcpy(P_PUBLIC_KEY_BUFADD, buffer_b);
                                                  PUBLIC_KEY_MAXLEN = 250;
                                                  PUBLIC_KEY_CURLEN = encodedSize_b;
```

```
                                                P_STATUS = 1; /*success*/
/*debug logic*/
#if ISaDEBUG
       fpb = fopen("debug-keygen-BIGNUM.txt", "w");
       fprintf(fpb, "PublicKeyBigNumber = ");
       BN_print_fp(fpb, PublicKeyBN);
       fprintf(fpb, " \n");
       fprintf(fpb, " \n");
       fprintf(fpb, "PrivateKeyBigNumber = ");
       BN_print_fp(fpb, PrivateKeyBN);
       fclose(fpb);

       fp = fopen("debug-keygen-Private.txt", "w");
       fprintf(fp, "hello world\n");
       fprintf(fp,"encodedSize = %i\n", encodedSize);
       fprintf(fp,"encodedSize_b = %i\n", encodedSize_b);
       fprintf(fp,"array dump = ");
       for (i = 0; i<250; i++)
       {
              fprintf(fp, "%c", PrivateKeyBinaryUnsignedArray[i]);
       }
       fprintf(fp, " \n");
       fprintf(fp,"PrivateKeyBinaryUnsigned = %s\n", PrivateKeyBinaryUnsigned);
       fprintf(fp,"PrivateKeyBase64 = %s\n", PrivateKeyBase64);
       fprintf(fp,"buffer = %s\n", buffer);
       fclose(fp);
#endif
                                          }
                                   }
                            }
                     }
              }
       }
   //END OF CUSTOM CODE

}

/* eof ********************************************************************/
```

A.3 – KeyVerify

```
/***************************************************************************
File:                keyverify.c
Author:              kfischer
Creation date:       15/06/2013 - 13:53
POU name:            KeyVerify
***************************************************************************/

---autocode---

//START OF CUSTOM CODE HEADERS
#include <debug.h>
#include <stdio.h>
#include <stdlib.h>
#include <openssl/ec.h>      // for EC_GROUP_new_by_curve_name, EC_GROUP_free,
EC_KEY_new, EC_KEY_set_group, EC_KEY_generate_key, EC_KEY_free
#include <openssl/ecdsa.h>   // for ECDSA_do_sign, ECDSA_do_verify
#include <openssl/obj_mac.h> // for NID_secp384r1
#include <openssl/bn.h>             // for BIGNUM conversions
#include <openssl/bio.h>
#include <openssl/evp.h>
#include <openssl/buffer.h>
#include <math.h>
#include <base64.h>                 // for base64 conversions
//END OF CUSTOM CODE HEADERS

---autocode---

   //START OF CUSTOM CODE


   //--Declarations

   /*debug variables*/
       #if ISaDEBUG
       FILE *fp;
       FILE *fpa;
       FILE *fpb;
       int i, j;
       #endif

   /*status variables*/
       #define P_PTR_PUBLIC_KEY_DATA (((char*)VA_ADDRESS(pvBfData,_P_IEC_DATA-
>public_key))+2)
       #define P_PTR_PRIVATE_KEY_DATA (((char*)VA_ADDRESS(pvBfData,_P_IEC_DATA-
>private_key))+2)

       const int set_group_success = 1;
       const int gen_success = 1;
       const int set_public_success = 1;
       const int set_private_success = 1;
       const int verify_success = 1;

       int set_group_status;
       int set_public_status;
       int set_private_status;
```

```
    int verify_status;

    unsigned char hash[] = "c7fbca202a95a570285e3d700eb04ca2";

/*base64 variables*/
    EC_KEY *eckey;
    EC_GROUP *ecgroup;
    EC_POINT *PublicKeyPoint;
    BIGNUM *PrivateKeyBN;
    BIGNUM *PublicKeyBN;
    ECDSA_SIG *signature;

    unsigned char *buffer;
    unsigned char *buffer_b;

    unsigned char *PublicKeyBinaryUnsigned, PublicKeyBinaryUnsignedArray[250];
    unsigned char *PrivateKeyBinaryUnsigned, PrivateKeyBinaryUnsignedArray[250];

    const unsigned char *PublicKeyBinaryConst;
    const unsigned char *PrivateKeyBinaryConst;



//--Logic
    P_STATUS = 0;

    memset(PrivateKeyBinaryUnsignedArray,0,250);
    PrivateKeyBinaryUnsigned = PrivateKeyBinaryUnsignedArray;

    memset(PublicKeyBinaryUnsignedArray,0,250);
    PublicKeyBinaryUnsigned = PublicKeyBinaryUnsignedArray;


    /*STEP 1 - create key object*/
    eckey=EC_KEY_new();
 if (NULL == eckey)
 {
     P_STATUS = -1;  /*failed to create key object*/
 }
 else
 {
           /*STEP 2 - create EC_GROUP object*/
           ecgroup = EC_GROUP_new_by_curve_name(NID_secp521r1);
     if (NULL == ecgroup)
     {
         P_STATUS = -2; /*failed to create new EC Group*/
     }
     else
     {
                  /*STEP 3 - associate key with group*/
         set_group_status = EC_KEY_set_group(eckey,ecgroup);
         if (set_group_success != set_group_status)
         {
                         P_STATUS = -3; /*failed to associate key with group*/
         }
         else
                  {
                         /*STEP 4 - set public key*/
                         PublicKeyPoint = EC_POINT_new(ecgroup);
```

```
                            PublicKeyBN = BN_new();

                            buffer_b = (unsigned char*) P_PTR_PUBLIC_KEY_DATA;

                            EVP_DecodeBlock(PublicKeyBinaryUnsigned,buffer_b,240);

                            PublicKeyBinaryConst = (const unsigned char*)
PublicKeyBinaryUnsigned;

                            BN_bin2bn(PublicKeyBinaryConst,133,PublicKeyBN);


       EC_POINT_bn2point(ecgroup,PublicKeyBN,PublicKeyPoint,NULL);


                            set_public_status =
EC_KEY_set_public_key(eckey,PublicKeyPoint);
                            if (set_public_success != set_public_status)
                            {
                                   P_STATUS = -4; /*failed to set public key*/
                            }
                            else
                            {
                                   /*STEP 5 - set private key*/
                                   PrivateKeyBN = BN_new();
                                   buffer = (unsigned char*) P_PTR_PRIVATE_KEY_DATA;

       EVP_DecodeBlock(PrivateKeyBinaryUnsigned,buffer,120);
                                   PrivateKeyBinaryConst = (const unsigned char*)
PrivateKeyBinaryUnsigned;

/*521 ECDSA is unique due to the extra bit resulting in a length of 65.125 bytes,
which means that when converting from bin to bn we will alternatively be converting 66
or 65 bytes depending on if the extra bit is a 1 or not.      The presence of the
extra bit is encoded in the in the first base64 characters. The byte 0000 0001 will be
broken in base64 to a pair of 0000000 and 1XXXXXX, which will result in the key having
a leading A character representing the 0000000.  However, the byte 0000 0000 will not
be encoded at all, which will result in the key not having a leading A.  Detection of
the leading A is therefore critical to properly convert the bin to BN.*/

                                   if (P_PTR_PRIVATE_KEY_DATA[0] == 'A')
                                   {

       BN_bin2bn(PrivateKeyBinaryConst,66,PrivateKeyBN);
                                   }
                                   else
                                   {

       BN_bin2bn(PrivateKeyBinaryConst,65,PrivateKeyBN);
                                   }

                                   set_private_status =
EC_KEY_set_private_key(eckey,PrivateKeyBN);
                                   if (set_private_success != set_private_status)
                                   {
                                          P_STATUS = -5; /*failed to set private key*/
                                   }
                                   else
                                   {
```

```
                                                /*STEP 6 - create signature on dummy hash*/
                                                signature = ECDSA_do_sign(hash,32,eckey);
                                                if (NULL == signature)
                                                {
                                                       P_STATUS = -6; /*failed to generate
signature*/
                                                }
                                                else
                                                {
                                                       /*STEP 7 - verify signature*/
                                                       verify_status =
ECDSA_do_verify(hash,32,signature,eckey);
                                                       if (verify_success != verify_status)
                                                       {
                                                              P_STATUS = -7; /*verification
failed*/
                                                       }
                                                       else
                                                       {
                                                              P_STATUS = 1; /*keys verified*/
                                                       }
                                                }
                                         }
                                  }
                           }
                    }
       }

       /*CLEANUP - Release memory structures to prevent memory leaks*/
       EC_KEY_free(eckey);                    /*frees EC_KEY memory allocation*/
       EC_GROUP_clear_free(ecgroup);   /*destroys any sensitive data in the group and
then frees the memory*/
       EC_POINT_free(PublicKeyPoint);  /*frees EC_POINT memory allocation*/
       BN_clear_free(PrivateKeyBN);    /*overwrites the BN before returning memory to
the system*/
       BN_clear_free(PublicKeyBN);
       ECDSA_SIG_free(signature);        /*frees ECDSA_SIG memory allocation*/


/*debug logic*/
#if ISaDEBUG
       fpa = fopen("debug-keyverify-public.txt", "w");
       fprintf(fpa, "hello world \n");
       fprintf(fpa, "P_PTR_PUBLIC_KEY_DATA = %s\n", P_PTR_PUBLIC_KEY_DATA);
       fprintf(fpa, " \n");
       fprintf(fpa, " \n");
       fprintf(fpa, "Public Key Array dump = ");
       for (j = 0; j<250; j++)
       {
              fprintf(fpa, "%c", PublicKeyBinaryUnsignedArray[j]);
       }
       fprintf(fpa, " \n");
       fprintf(fpa, " \n");
       fclose(fpa);

       fp = fopen("debug-keyverify-private.txt", "w");
       fprintf(fp, "hello world\n");
       fprintf(fp, "P_PTR_PRIVATE_KEY_DATA = %s\n", P_PTR_PRIVATE_KEY_DATA);
       fprintf(fp, "P_PTR_PRIVATE_KEY_DATA[0] = %c\n", P_PTR_PRIVATE_KEY_DATA[0]);
```

```
        fprintf(fp, "Private Key Array dump = ");
        for (i = 0; i<250; i++)
        {
               fprintf(fp, "%c", PrivateKeyBinaryUnsignedArray[i]);
        }
        fprintf(fp, " \n");
        fclose(fp);


        fpb = fopen("debug-keyverify-BIGNUM.txt", "w");
        fprintf(fpb, "PublicKeyBigNumber = ");
        BN_print_fp(fpb, PublicKeyBN);
        fprintf(fpb, " \n");
        fprintf(fpb, " \n");
        fprintf(fpb, "PrivateKeyBigNumber = ");
        BN_print_fp(fpb, PrivateKeyBN);
        fprintf(fpb, " \n");
        fprintf(fpb, " \n");
        fprintf(fpb, "bnlen = %i\n", bnlen);
        fclose(fpb);
#endif




   //END OF CUSTOM CODE


}

/* eof *******************************************************************/
```

A.4 – SigGen

```
/***************************************************************************
File:               siggen.c
Author:             kfischer
Creation date:      10/07/2013 - 22:27
POU name:           SigGen
***************************************************************************/

---autocode---

//START OF CUSTOM CODE HEADERS
#include <debug.h>
#include <stdio.h>
#include <stdlib.h>
#include <openssl/ec.h>      // for EC_GROUP_new_by_curve_name, EC_GROUP_free,
EC_KEY_new, EC_KEY_set_group, EC_KEY_generate_key, EC_KEY_free
#include <openssl/ecdsa.h>   // for ECDSA_do_sign, ECDSA_do_verify
#include <openssl/obj_mac.h> // for NID_secp384r1
#include <openssl/bn.h>             // for BIGNUM conversions
#include <openssl/bio.h>
#include <openssl/evp.h>
#include <openssl/buffer.h>
#include <math.h>
#include <base64.h>                 // for base64 conversions
//END OF CUSTOM CODE HEADERS

---autocode---

   //START OF CUSTOM CODE


   //--Declarations

   /*debug variables*/
       #if ISaDEBUG
       FILE *fp;
       int i;
       #endif

   /*status variables*/

       #define P_PTR_PRIVATE_KEY_DATA (((char*)VA_ADDRESS(pvBfData,_P_IEC_DATA-
>private_key))+2)
       #define P_PTR_HASH_DATA (((char*)VA_ADDRESS(pvBfData,_P_IEC_DATA->hash))+2)

       const int set_group_success = 1;
       const int gen_success = 1;
       const int set_public_success = 1;
       const int set_private_success = 1;
       const int verify_success = 1;

       int set_group_status;
       int set_private_status;


       unsigned char hash[128];
```

```
/*EC Variables*/
    EC_KEY *eckey;
    EC_GROUP *ecgroup;
    ECDSA_SIG *signature;

/*base64 variables for decomposing private key*/
    BIGNUM *PrivateKeyBN;
    unsigned char *buffer;
 unsigned char *PrivateKeyBinaryUnsigned, PrivateKeyBinaryUnsignedArray[250];
    const unsigned char *PrivateKeyBinaryConst;

/*base64 variables for encoding SIG_R and SIG_S*/

    int SIG_R_EncodedSize;
    unsigned char *SIG_R_BinaryUnsigned, SIG_R_BinaryUnsignedArray[250];
 unsigned char *SIG_R_Base64, SIG_R_Base64Array[250];
    int SIG_S_EncodedSize;
    unsigned char *SIG_S_BinaryUnsigned, SIG_S_BinaryUnsignedArray[250];
 unsigned char *SIG_S_Base64, SIG_S_Base64Array[250];

    char *SIG_R_Buffer, SIG_R_BufferArray[250];
    char *SIG_S_Buffer, SIG_S_BufferArray[250];

//--Logic

    /*STEP 0 - Initialization*/
    P_STATUS = 0;

    memcpy(hash,P_PTR_HASH_DATA,128);

    memset(PrivateKeyBinaryUnsignedArray,0,250);
    PrivateKeyBinaryUnsigned = PrivateKeyBinaryUnsignedArray;

    memset(SIG_R_BinaryUnsignedArray,0,250);
    SIG_R_BinaryUnsigned = SIG_R_BinaryUnsignedArray;
    memset(SIG_R_Base64Array,0,250);
    SIG_R_Base64 = SIG_R_Base64Array;
    memset(SIG_R_BufferArray,0,250);
    SIG_R_Buffer = SIG_R_BufferArray;

    memset(SIG_S_BinaryUnsignedArray,0,250);
    SIG_S_BinaryUnsigned = SIG_S_BinaryUnsignedArray;
    memset(SIG_S_Base64Array,0,250);
    SIG_S_Base64 = SIG_S_Base64Array;
    memset(SIG_S_BufferArray,0,250);
    SIG_S_Buffer = SIG_S_BufferArray;

    /*STEP 1 - create key object*/
    eckey=EC_KEY_new();
 if (NULL == eckey)
 {
     P_STATUS = -1;  /*failed to create key object*/
 }
 else
 {
           /*STEP 2 - create EC_GROUP object*/
           ecgroup = EC_GROUP_new_by_curve_name(NID_secp521r1);
     if (NULL == ecgroup)
     {
            P_STATUS = -2; /*failed to create new EC Group*/
        }
        else
        {
                    /*STEP 3 - associate key with group*/
            set_group_status = EC_KEY_set_group(eckey,ecgroup);
            if (set_group_success != set_group_status)
            {
                        P_STATUS = -3; /*failed to associate key with group*/
            }
            else
                    {
                        /*STEP 4 - set private key*/
                        PrivateKeyBN = BN_new();

                        buffer = (unsigned char*) P_PTR_PRIVATE_KEY_DATA;
                        EVP_DecodeBlock(PrivateKeyBinaryUnsigned,buffer,120);

                        PrivateKeyBinaryConst = (const unsigned char*)
PrivateKeyBinaryUnsigned;

/*521 ECDSA is unique due to the extra bit resulting in a length of 65.125 bytes,
which means that when converting from bin to bn we will alternatively be converting 66
or 65 bytes depending on if the extra bit is a 1 or not.      The presence of the
extra bit is encoded in the in the first  base64 characters. The byte 0000 0001 will be
broken in base64 to a pair of 0000000 and 1XXXXXX, which will result in the key having
a leading A character representing the 0000000.  However, the byte 0000 0000 will not
be encoded at all, which will result in the key not having a leading A.  Detection of
the leading A is therefore critical to properly convert the bin to BN.*/

                        if (P_PTR_PRIVATE_KEY_DATA[0] == 'A')
                        {
                              BN_bin2bn(PrivateKeyBinaryConst,66,PrivateKeyBN);
                        }
                        else
                        {
                              BN_bin2bn(PrivateKeyBinaryConst,65,PrivateKeyBN);
                        }

                        set_private_status =
EC_KEY_set_private_key(eckey,PrivateKeyBN);
                        if (set_private_success != set_private_status)
                        {
                              P_STATUS = -4; /*failed to set private key*/
                        }
                        else
                        {
                              /*STEP 5 - create signature on dummy hash*/
                              signature = ECDSA_do_sign(hash,128,eckey);
                              if (NULL == signature)
                              {
                                    P_STATUS = -5; /*failed to generate
signature*/
                              }
                              else
                              {
                                    /*STEP 6 - base64 encode SIG_R and SIG_S and
output result*/
```

```
                                          BN_bn2bin(signature->r,SIG_R_BinaryUnsigned);
                                          SIG_R_EncodedSize =
EVP_EncodeBlock(SIG_R_Base64,SIG_R_BinaryUnsigned,89);

                                          BN_bn2bin(signature->s,SIG_S_BinaryUnsigned);
                                          SIG_S_EncodedSize =
EVP_EncodeBlock(SIG_S_Base64,SIG_S_BinaryUnsigned,89);

                                          SIG_R_Buffer = (char *) SIG_R_Base64;
                                          SIG_S_Buffer = (char *) SIG_S_Base64;

                                          strcpy(P_SIG_R_BUFADD,SIG_R_Buffer);
                                          SIG_R_MAXLEN = 250;
                                          SIG_R_CURLEN = SIG_R_EncodedSize;

                                          strcpy(P_SIG_S_BUFADD,SIG_S_Buffer);
                                          SIG_S_MAXLEN = 250;
                                          SIG_S_CURLEN = SIG_S_EncodedSize;

                                          P_STATUS = 1; /*success*/
                                   }
                            }
                     }
              }
       }

       ///*CLEANUP - Release memory structures to prevent memory leaks*/
       EC_KEY_free(eckey);                      /*frees EC_KEY memory allocation*/
       EC_GROUP_clear_free(ecgroup);   /*destroys any sensitive data in the group and
then frees the memory*/
       BN_clear_free(PrivateKeyBN);    /*overwrites the BN before returning memory to
the system*/
       ECDSA_SIG_free(signature);         /*frees ECDSA_SIG memory allocation*/


/*debug logic*/
#if ISaDEBUG
       fp = fopen("debug-siggen.txt", "w");
       fprintf(fp, "hello world\n");
       fprintf(fp, " \n");
       fprintf(fp, "PrivateKeyBigNumber = ");
       BN_print_fp(fp, PrivateKeyBN);
       fprintf(fp, " \n");
       fprintf(fp, "signature->r = ");
       BN_print_fp(fp, signature->r);
       fprintf(fp, " \n");
       fprintf(fp, "signature->s = ");
       BN_print_fp(fp, signature->s);
       fprintf(fp, " \n");
       fclose(fp);

#endif

   //END OF CUSTOM CODE

}

/* eof *******************************************************************/
```

A.5 – SigVerify

```
/***************************************************************************
File:               sigverify.c
Author:             kfischer
Creation date:      12/07/2013 - 18:52
POU name:           SigVerify
****************************************************************************/

---autocode---

//START OF CUSTOM CODE HEADERS
#include <debug.h>
#include <stdio.h>
#include <stdlib.h>
#include <openssl/ec.h>      // for EC_GROUP_new_by_curve_name, EC_GROUP_free,
EC_KEY_new, EC_KEY_set_group, EC_KEY_generate_key, EC_KEY_free
#include <openssl/ecdsa.h>   // for ECDSA_do_sign, ECDSA_do_verify
#include <openssl/obj_mac.h> // for NID_secp384r1
#include <openssl/bn.h>             // for BIGNUM conversions
#include <openssl/bio.h>
#include <openssl/evp.h>
#include <openssl/buffer.h>
#include <math.h>
#include <base64.h>                 // for base64 conversions
//END OF CUSTOM CODE HEADERS

---autocode---

   //START OF CUSTOM CODE


   //--Declarations

   /*debug variables*/
       #if ISaDEBUG
       FILE *fp;
       int i;
       #endif

       /*status variables*/
       #define P_PTR_PUBLIC_KEY_DATA (((char*)VA_ADDRESS(pvBfData,_P_IEC_DATA-
>public_key))+2)
       #define P_PTR_HASH_DATA (((char*)VA_ADDRESS(pvBfData,_P_IEC_DATA->hash))+2)
       #define P_PTR_SIG_R_DATA (((char*)VA_ADDRESS(pvBfData,_P_IEC_DATA->sig_r))+2)
       #define P_PTR_SIG_S_DATA (((char*)VA_ADDRESS(pvBfData,_P_IEC_DATA->sig_s))+2)

       const int set_group_success = 1;
       const int gen_success = 1;
       const int set_public_success = 1;
       const int set_private_success = 1;
       const int verify_success = 1;

       int set_group_status;
       int set_public_status;
       int verify_status;

       unsigned char hash[128];
```

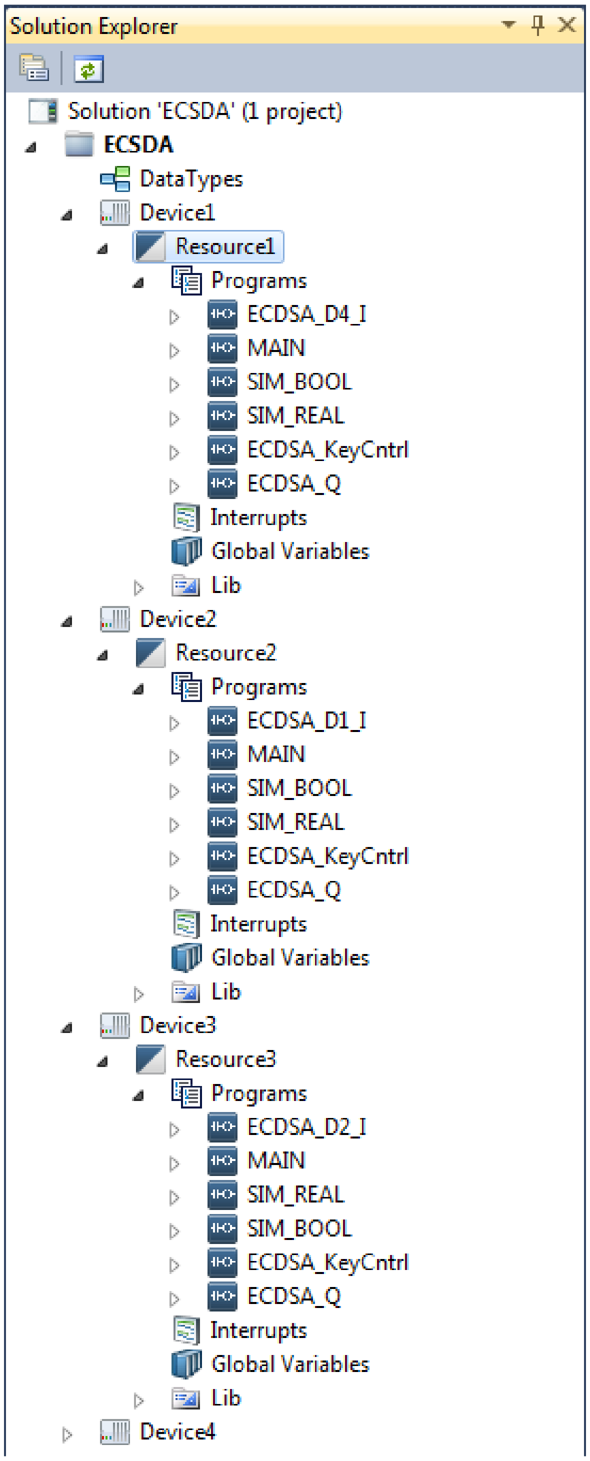
```
 /*EC Variables*/
    EC_KEY *eckey;
    EC_GROUP *ecgroup;
    EC_POINT *PublicKeyPoint;
    ECDSA_SIG *signature;


 /*base64 variables for decoding public key*/
    BIGNUM *PublicKeyBN;
    unsigned char *buffer;
 unsigned char *PublicKeyBinaryUnsigned, PublicKeyBinaryUnsignedArray[250];
    const unsigned char *PublicKeyBinaryConst;

    /*base64 variables for decoding SIG_R key*/
    BIGNUM *SIG_R_BN;
    unsigned char *SIG_R_buffer;
 unsigned char *SIG_R_BinaryUnsigned, SIG_R_BinaryUnsignedArray[250];
    const unsigned char *SIG_R_BinaryConst;

    /*base64 variables for decoding SIG_S key*/
    BIGNUM *SIG_S_BN;
    unsigned char *SIG_S_buffer;
 unsigned char *SIG_S_BinaryUnsigned, SIG_S_BinaryUnsignedArray[250];
    const unsigned char *SIG_S_BinaryConst;

//--Logic
    P_STATUS = 0;

    memcpy(hash,P_PTR_HASH_DATA,128);

    memset(PublicKeyBinaryUnsignedArray,0,250);
    PublicKeyBinaryUnsigned = PublicKeyBinaryUnsignedArray;

    memset(SIG_R_BinaryUnsignedArray,0,250);
    SIG_R_BinaryUnsigned = SIG_R_BinaryUnsignedArray;

    memset(SIG_S_BinaryUnsignedArray,0,250);
    SIG_S_BinaryUnsigned = SIG_S_BinaryUnsignedArray;

    /*STEP 1 - create key object*/
    eckey=EC_KEY_new();
 if (NULL == eckey)
 {
     P_STATUS = -1;  /*failed to create key object*/
 }
 else
 {
           /*STEP 2 - create EC_GROUP object*/
           ecgroup = EC_GROUP_new_by_curve_name(NID_secp521r1);
     if (NULL == ecgroup)
     {
         P_STATUS = -2; /*failed to create new EC Group*/
     }
     else
     {
                  /*STEP 3 - associate key with group*/
         set_group_status = EC_KEY_set_group(eckey,ecgroup);
         if (set_group_success != set_group_status)
         {
```

```
                            P_STATUS = -3; /*failed to associate key with group*/
            }
            else
                    {
                            /*STEP 4 - set public key*/
                            PublicKeyPoint = EC_POINT_new(ecgroup);
                            PublicKeyBN = BN_new();
                            buffer = (unsigned char*) P_PTR_PUBLIC_KEY_DATA;
                            EVP_DecodeBlock(PublicKeyBinaryUnsigned,buffer,240);
                            PublicKeyBinaryConst = (const unsigned char*)
PublicKeyBinaryUnsigned;
                            BN_bin2bn(PublicKeyBinaryConst,133,PublicKeyBN);


       EC_POINT_bn2point(ecgroup,PublicKeyBN,PublicKeyPoint,NULL);
                            set_public_status =
EC_KEY_set_public_key(eckey,PublicKeyPoint);
                            if (set_public_success != set_public_status)
                            {
                                   P_STATUS = -4; /*failed to set public key*/
                            }
                            else
                            {
                                   /*STEP 5 - convert Base64 SIG_R and SIG_S into BN
and set signature values*/

                                   signature = ECDSA_SIG_new();
                                   SIG_R_BN = signature->r;
                                   SIG_S_BN = signature->s;

                                   SIG_R_buffer = (unsigned char*) P_PTR_SIG_R_DATA;

       EVP_DecodeBlock(SIG_R_BinaryUnsigned,SIG_R_buffer,120);
                                   SIG_R_BinaryConst = (const unsigned char*)
SIG_R_BinaryUnsigned;
                                   if (P_PTR_SIG_R_DATA[0] == 'A')
                                   {
                                          BN_bin2bn(SIG_R_BinaryConst,66,SIG_R_BN);
                                   }
                                   else
                                   {
                                          BN_bin2bn(SIG_R_BinaryConst,65,SIG_R_BN);
                                   }

                                   SIG_S_buffer = (unsigned char*) P_PTR_SIG_S_DATA;

       EVP_DecodeBlock(SIG_S_BinaryUnsigned,SIG_S_buffer,120);
                                   SIG_S_BinaryConst = (const unsigned char*)
SIG_S_BinaryUnsigned;
                                   if (P_PTR_SIG_S_DATA[0] == 'A')
                                   {
                                          BN_bin2bn(SIG_S_BinaryConst,66,SIG_S_BN);
                                   }
                                   else
                                   {
                                          BN_bin2bn(SIG_S_BinaryConst,65,SIG_S_BN);
                                   }

                                   if (NULL==signature)
```

```
                                    {
                                           P_STATUS = -5;
                                    }
                                    else
                                    {
                                           /*STEP 6 - Verify Signature*/
                                           verify_status = 
ECDSA_do_verify(hash,128,signature,eckey);
                                           if (verify_success != verify_status)
                                           {
                                                  P_STATUS = -6; /*verification failed*/
                                           }
                                           else
                                           {
                                                  P_STATUS = 1; /*signature verified*/
                                           }
                                    }
                             }
                      }
               }
       }

       /*CLEANUP - Release memory structures to prevent memory leaks*/

              EC_KEY_free(eckey);                     /*frees EC_KEY memory 
allocation*/
              EC_GROUP_clear_free(ecgroup);   /*destroys any sensitive data in the 
group and then frees the memory*/
              EC_POINT_free(PublicKeyPoint);  /*frees EC_POINT memory allocation*/
              BN_clear_free(PublicKeyBN);    /*overwrites the BN before returning 
memory to the system*/
              ECDSA_SIG_free(signature);        /*frees ECDSA_SIG memory allocation*/



/*debug logic*/
#if ISaDEBUG
       fp = fopen("debug-sigverify.txt", "w");
       fprintf(fp, "hello world\n");
       fprintf(fp, " \n");
       fprintf(fp, "PublicKeyBigNumber = ");
       BN_print_fp(fp, PublicKeyBN);
       fprintf(fp, " \n");
       fprintf(fp, "signature->r = ");
       BN_print_fp(fp, signature->r);
       fprintf(fp, " \n");
       fprintf(fp, "signature->s = ");
       BN_print_fp(fp, signature->s);
       fprintf(fp, " \n");
       fclose(fp);

#endif



}

/* eof ********************************************************************/
```

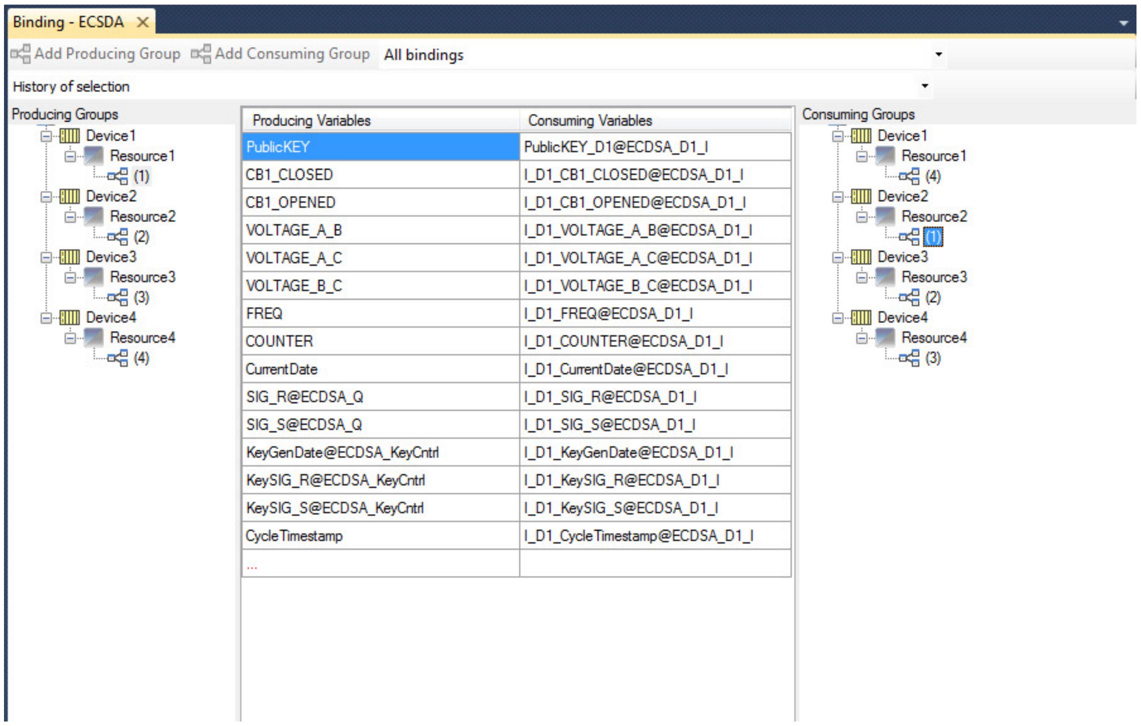
APPENDIX B – LADDER LOGIC CODE

B.1 – Solution Explorer



```
Solution 'ECSDA' (1 project)
  ECSDA
    DataTypes
    Device1
      Resource1
        Programs
          ECDSA_D4_I
          MAIN
          SIM_BOOL
          SIM_REAL
          ECDSA_KeyCntrl
          ECDSA_Q
        Interrupts
        Global Variables
        Lib
    Device2
      Resource2
        Programs
          ECDSA_D1_I
          MAIN
          SIM_BOOL
          SIM_REAL
          ECDSA_KeyCntrl
          ECDSA_Q
        Interrupts
        Global Variables
        Lib
    Device3
      Resource3
        Programs
          ECDSA_D2_I
          MAIN
          SIM_REAL
          SIM_BOOL
          ECDSA_KeyCntrl
          ECDSA_Q
        Interrupts
        Global Variables
        Lib
    Device4
```

B.2 – Bindings Sample


Binding - ECSDA

Add Producing Group | Add Consuming Group | All bindings

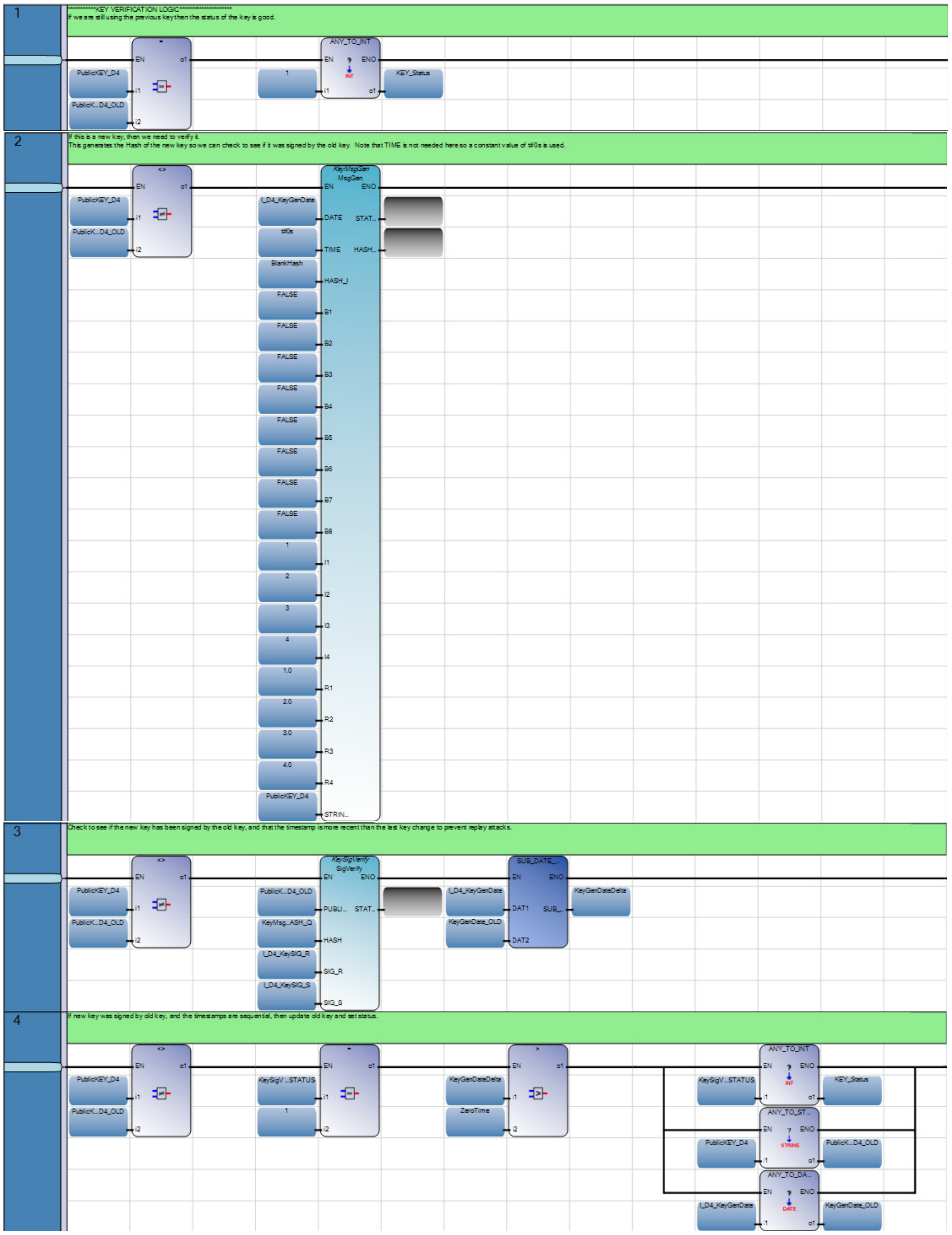
History of selection

Producing Groups
- Device1
  - Resource1
    - (1)
- Device2
  - Resource2
    - (2)
- Device3
  - Resource3
    - (3)
- Device4
  - Resource4
    - (4)

Producing Variables	Consuming Variables
PublicKEY	PublicKEY_D1@ECDSA_D1_I
CB1_CLOSED	I_D1_CB1_CLOSED@ECDSA_D1_I
CB1_OPENED	I_D1_CB1_OPENED@ECDSA_D1_I
VOLTAGE_A_B	I_D1_VOLTAGE_A_B@ECDSA_D1_I
VOLTAGE_A_C	I_D1_VOLTAGE_A_C@ECDSA_D1_I
VOLTAGE_B_C	I_D1_VOLTAGE_B_C@ECDSA_D1_I
FREQ	I_D1_FREQ@ECDSA_D1_I
COUNTER	I_D1_COUNTER@ECDSA_D1_I
CurrentDate	I_D1_CurrentDate@ECDSA_D1_I
SIG_R@ECDSA_Q	I_D1_SIG_R@ECDSA_D1_I
SIG_S@ECDSA_Q	I_D1_SIG_S@ECDSA_D1_I
KeyGenDate@ECDSA_KeyCntrl	I_D1_KeyGenDate@ECDSA_D1_I
KeySIG_R@ECDSA_KeyCntrl	I_D1_KeySIG_R@ECDSA_D1_I
KeySIG_S@ECDSA_KeyCntrl	I_D1_KeySIG_S@ECDSA_D1_I
CycleTimestamp	I_D1_CycleTimestamp@ECDSA_D1_I
...	

Consuming Groups
- Device1
  - Resource1
    - (4)
- Device2
  - Resource2
    - (1)
- Device3
  - Resource3
    - (2)
- Device4
  - Resource4
    - (3)

B.3 – ECDSA_D4_I

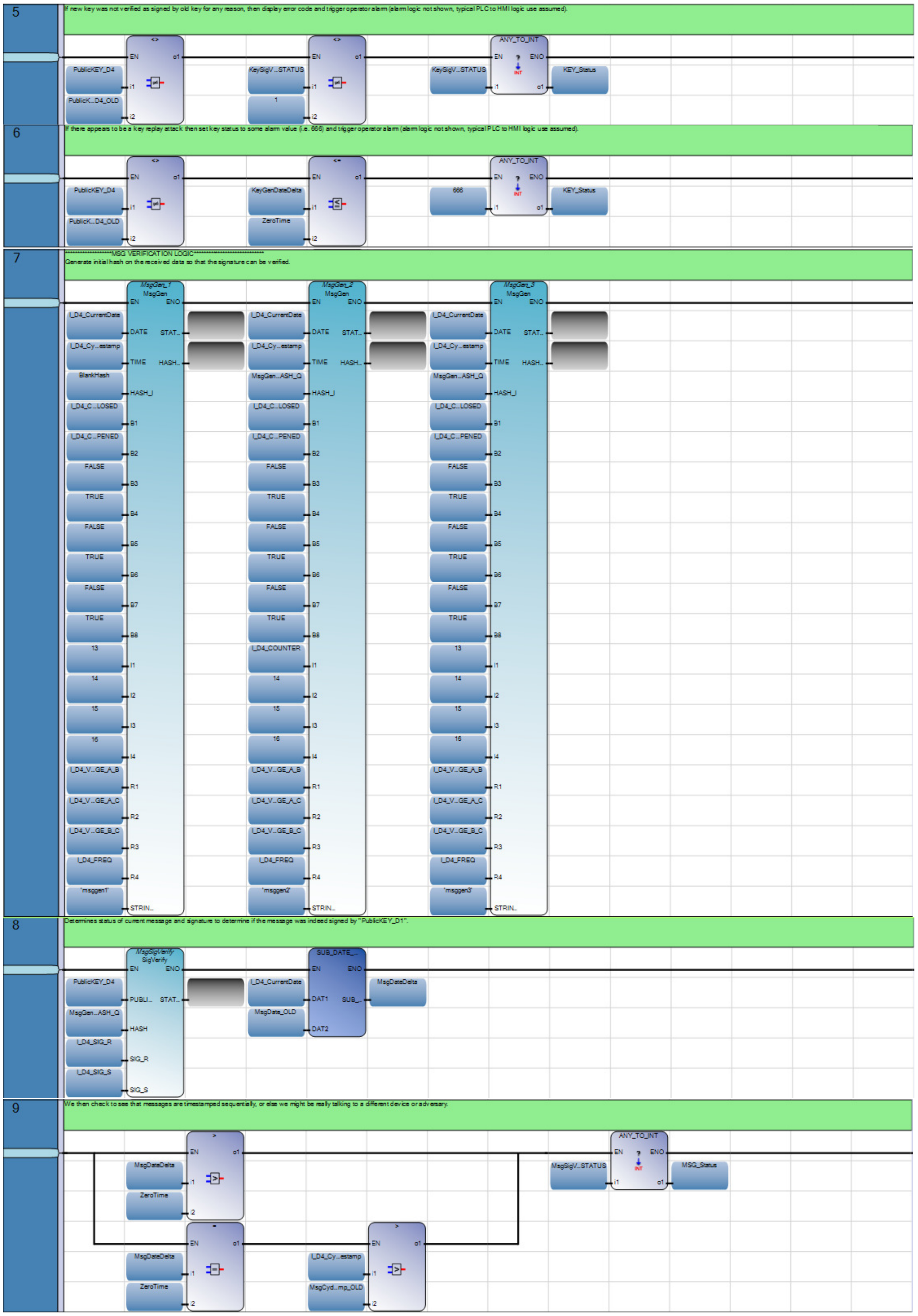


1
************KEY VERIFICATION LOGIC**********************
If we are still using the previous key then the status of the key is good.

2
If this is a new key, then we need to verify it.
This generates the Hash of the new key so we can check to see if it was signed by the old key. Note that TIME is not needed here so a constant value of t#0s is used.

3
Check to see if the new key has been signed by the old key, and that the timestamp is more recent than the last key change to prevent replay attacks.

4
If new key was signed by old key, and the timestamps are sequential, then update old key and set status.

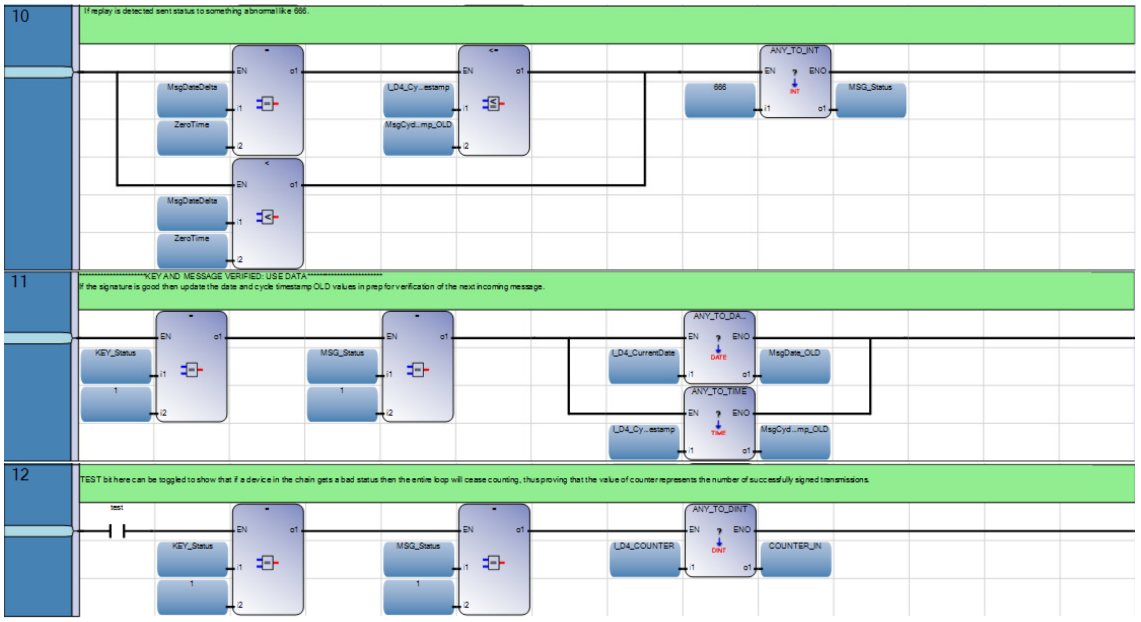

5 If new key was not verified as signed by old key for any reason, then display error code and trigger operator alarm (alarm logic not shown, typical PLC to HMI logic use assumed).

6 If there appears to be a key replay attack then set key status to some alarm value (i.e. 666) and trigger operator alarm (alarm logic not shown, typical PLC to HMI logic use assumed).

7 ********************MSG VERIFICATION LOGIC*******************************
Generate initial hash on the received data so that the signature can be verified.

8 Determines status of current message and signature to determine if the message was indeed signed by "PublicKEY_D1".

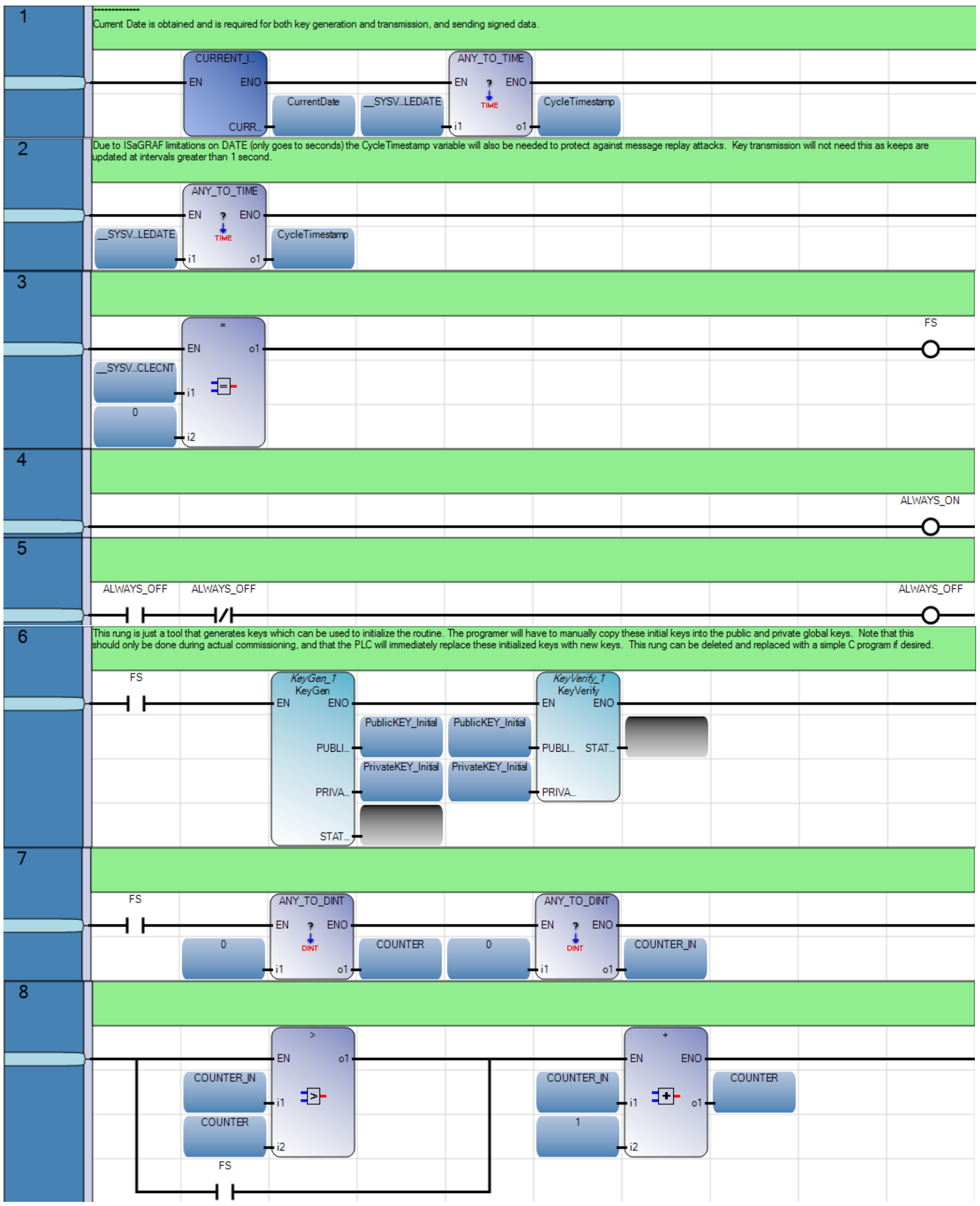
9 We then check to see that messages are timestamped sequentially, or else we might be really talking to a different device or adversary.


10 If replay is detected sent status to something abnormal like 666.

11 ********************KEY AND MESSAGE VERIFIED: USE DATA************************
If the signature is good then update the date and cycle timestamp OLD values in prep for verification of the next incoming message.

12 TEST bit here can be toggled to show that if a device in the chain gets a bad status then the entire loop will cease counting, thus proving that the value of counter represents the number of successfully signed transmissions.

B.4 – MAIN

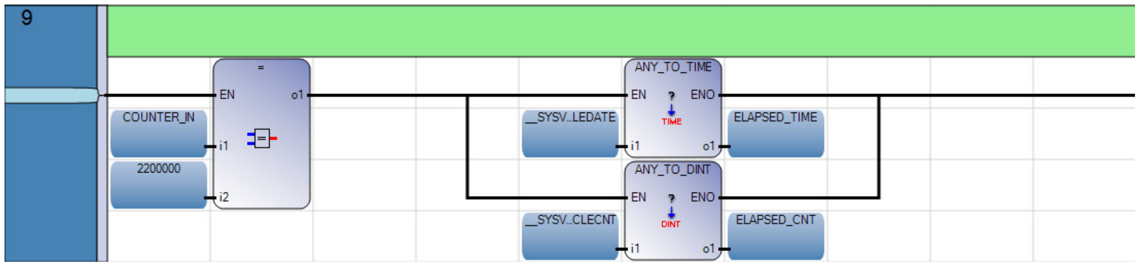


****

Current Date is obtained and is required for both key generation and transmission, and sending signed data.

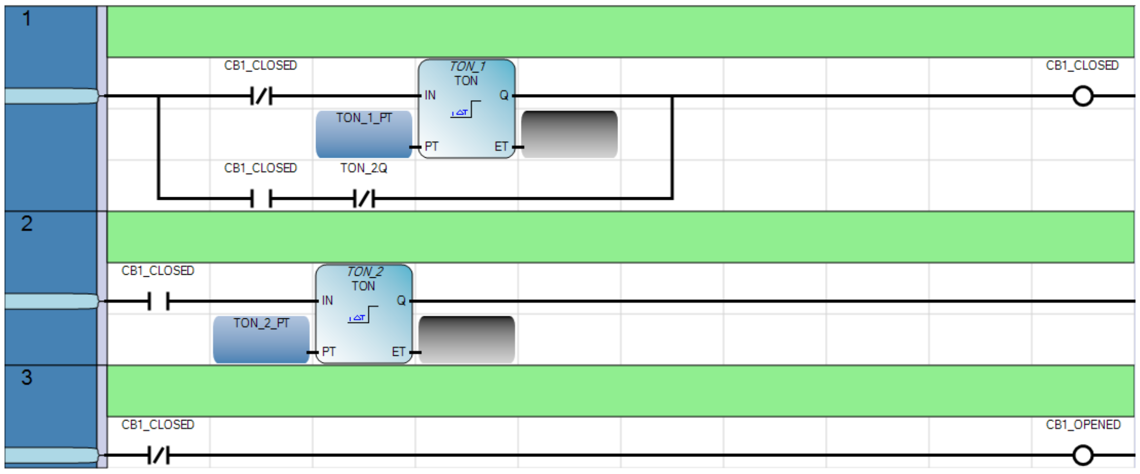
Due to ISaGRAF limitations on DATE (only goes to seconds) the CycleTimestamp variable will also be needed to protect against message replay attacks. Key transmission will not need this as keeps are updated at intervals greater than 1 second.

This rung is just a tool that generates keys which can be used to initialize the routine. The programer will have to manually copy these initial keys into the public and private global keys. Note that this should only be done during actual commissioning, and that the PLC will immediately replace these initialized keys with new keys. This rung can be deleted and replaced with a simple C program if desired.

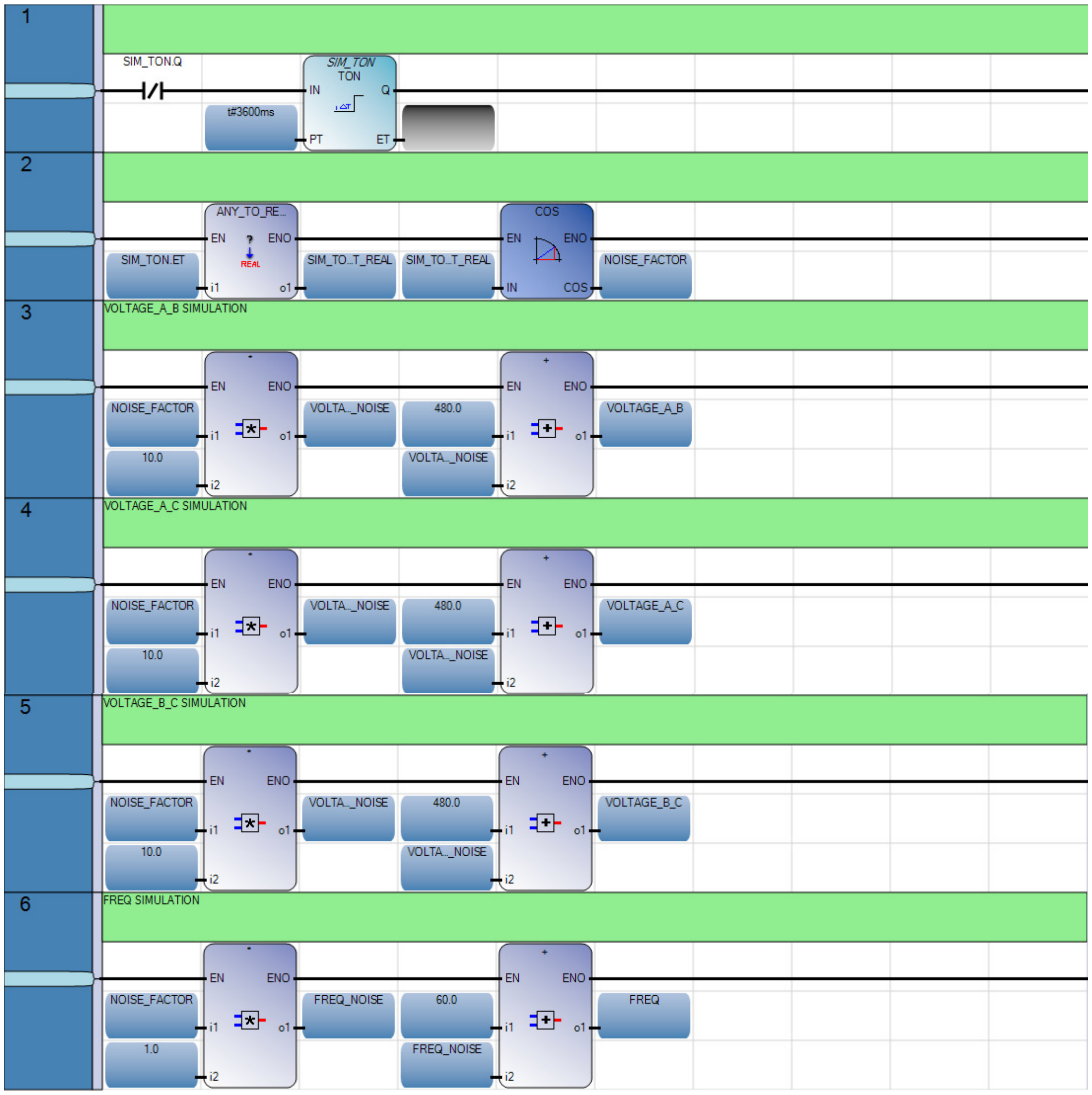


Note: Rungs 8 and 9 are only used for the main routine of Device 1 to initialize the device and to trap the number elapsed time and cycle counts. The other devices have a rung similar to rung 8 but without the FS XIC latch.

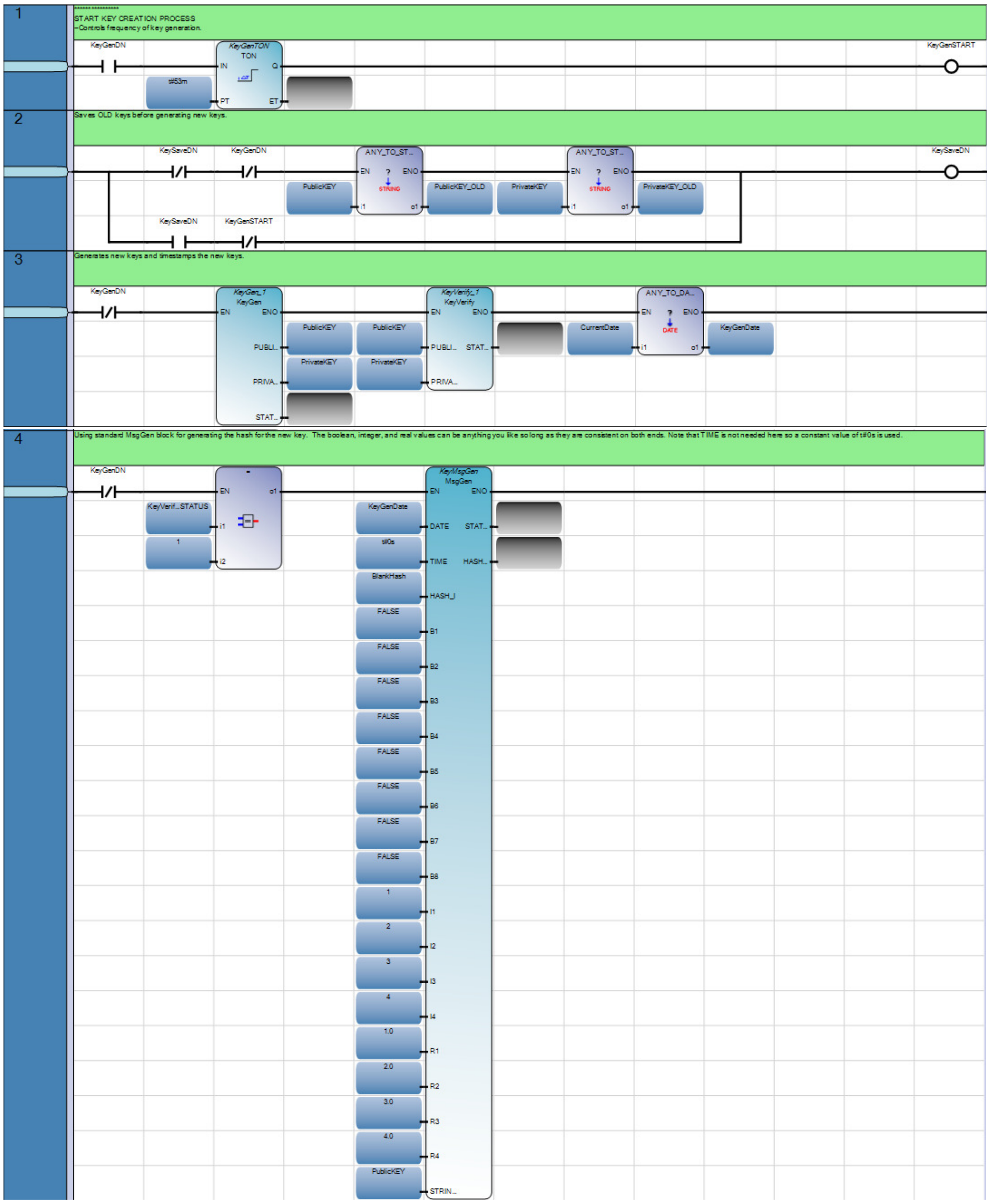
B.5 – SIM_BOOL

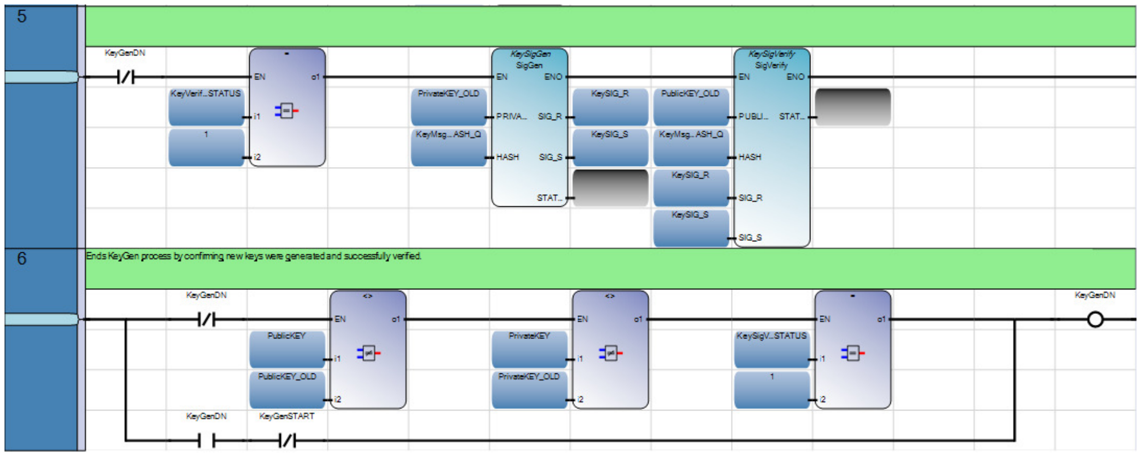

1
CB1_CLOSED
TON_1
TON
IN
Q
TON_1_PT
PT
ET
CB1_CLOSED
CB1_CLOSED
TON_2Q
2
CB1_CLOSED
TON_2
TON
IN
Q
TON_2_PT
PT
ET
3
CB1_CLOSED
CB1_OPENED

B.6 – SIM_REAL


1
SIM_TON.Q
SIM_TON
TON
IN
Q
t#3600ms
PT
ET
2
ANY_TO_RE...
EN
ENO
REAL
SIM_TON.ET
i1
o1
SIM_TO..T_REAL
SIM_TO..T_REAL
COS
EN
ENO
IN
COS
NOISE_FACTOR
3
VOLTAGE_A_B SIMULATION
NOISE_FACTOR
10.0
VOLTA..._NOISE
480.0
VOLTA..._NOISE
VOLTAGE_A_B
4
VOLTAGE_A_C SIMULATION
NOISE_FACTOR
10.0
VOLTA..._NOISE
480.0
VOLTA..._NOISE
VOLTAGE_A_C
5
VOLTAGE_B_C SIMULATION
NOISE_FACTOR
10.0
VOLTA..._NOISE
480.0
VOLTA..._NOISE
VOLTAGE_B_C
6
FREQ SIMULATION
NOISE_FACTOR
1.0
FREQ_NOISE
60.0
FREQ_NOISE
FREQ

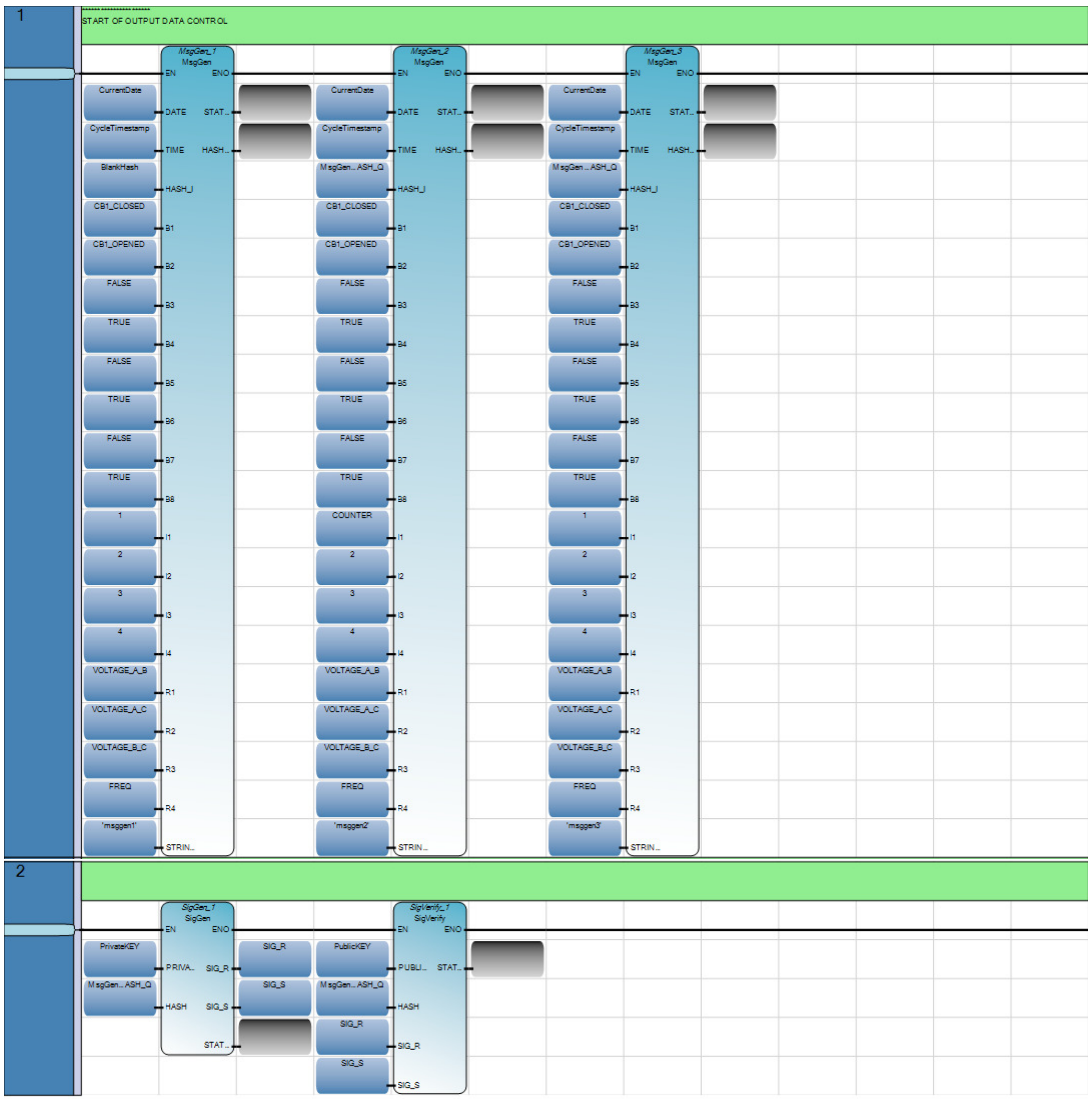
B.7 – ECDSA_KeyCntrl





Ends KeyGen process by confirming new keys were generated and successfully verified.

B.8 – ECDSA_Q


1
START OF OUTPUT DATA CONTROL
MsgGen_1
MsgGen
EN
ENO
CurrentDate
DATE
STAT...
CycleTimestamp
TIME
HASH...
BlankHash
HASH_I
CB1_CLOSED
B1
CB1_OPENED
B2
FALSE
B3
TRUE
B4
FALSE
B5
TRUE
B6
FALSE
B7
TRUE
B8
1
I1
2
I2
3
I3
4
I4
VOLTAGE_A_B
R1
VOLTAGE_A_C
R2
VOLTAGE_B_C
R3
FREQ
R4
'msggen1'
STRIN...
MsgGen_2
MsgGen
EN
ENO
CurrentDate
DATE
STAT...
CycleTimestamp
TIME
HASH...
MsgGen...ASH_Q
HASH_I
CB1_CLOSED
B1
CB1_OPENED
B2
FALSE
B3
TRUE
B4
FALSE
B5
TRUE
B6
FALSE
B7
TRUE
B8
COUNTER
I1
2
I2
3
I3
4
I4
VOLTAGE_A_B
R1
VOLTAGE_A_C
R2
VOLTAGE_B_C
R3
FREQ
R4
'msggen2'
STRIN...
MsgGen_3
MsgGen
EN
ENO
CurrentDate
DATE
STAT...
CycleTimestamp
TIME
HASH...
MsgGen...ASH_Q
HASH_I
CB1_CLOSED
B1
CB1_OPENED
B2
FALSE
B3
TRUE
B4
FALSE
B5
TRUE
B6
FALSE
B7
TRUE
B8
1
I1
2
I2
3
I3
4
I4
VOLTAGE_A_B
R1
VOLTAGE_A_C
R2
VOLTAGE_B_C
R3
FREQ
R4
'msggen3'
STRIN...
2
SigGen_1
SigGen
EN
ENO
PrivateKEY
PRIVA...
SIG_R
SIG_R
MsgGen...ASH_Q
HASH
SIG_S
SIG_S
STAT...
SigVerify_1
SigVerify
EN
ENO
PublicKEY
PUBLI...
STAT...
MsgGen...ASH_Q
HASH
SIG_R
SIG_R
SIG_S
SIG_S