I. PLCrypto Manual for Application Developer

PLCrypto is developed base on structured text (ST) and Allen Bradley's Studio 5000 (30 version). So it can be directly run over Allen Bradley's PLCs. PLCrypto includes the most widely used (symmetric） lightweight cryptographic algorithms, such as one-way function (OWF), universal one-way hash function (UOWHF), message authentication code (CHASKEY), block ciphers (PRESENT, SPECK, and SIMON), and pseudo-random function/generator (PRF/PRG), collision-resistant hash functions (PHOTON and SPONGENT), big integer operations, and a protocol called proof of aliveness.

*__Note__: if users want to run PLCrypto on other platform, they can copy the ST codes to the target platforms that support **IEC 61131-3** compliant ST.

II. Algorithms in PLCrypto

1. One-way function (OWF)

(1) Open the project improved_OWF_HardCode_IF.ACD;

(2) Init DINT variable **OWF_Ver** to be j 2 f0; 1; 2g to specify the version of OWF, where '0' denotes 256-bit input/output, '1' denotes 384-bit input/ output, and '2' denotes 512-bit input/output;

(3) Assign the input values to tags f OWF_Input[i]g$_{i2[NumDINT]}$ where NumDINT 2 f8; 12; 16g are determined by **OWF_Ver**, e.g., 8*32=256 bits;

(4) Call **Improved_OWF**, via **JSR(Improved_OWF),** to run the OWF Evaluation;

(5) Get the results in DINT ARRAY f OWF_Result[i]g$_{i2[NumDINT]}$where NumDINT 2 f9; 13; 17g corresponds to specific versions, respectively. Each DINT of results has 31-bit.

In Figure1, we show an example code of the above steps.

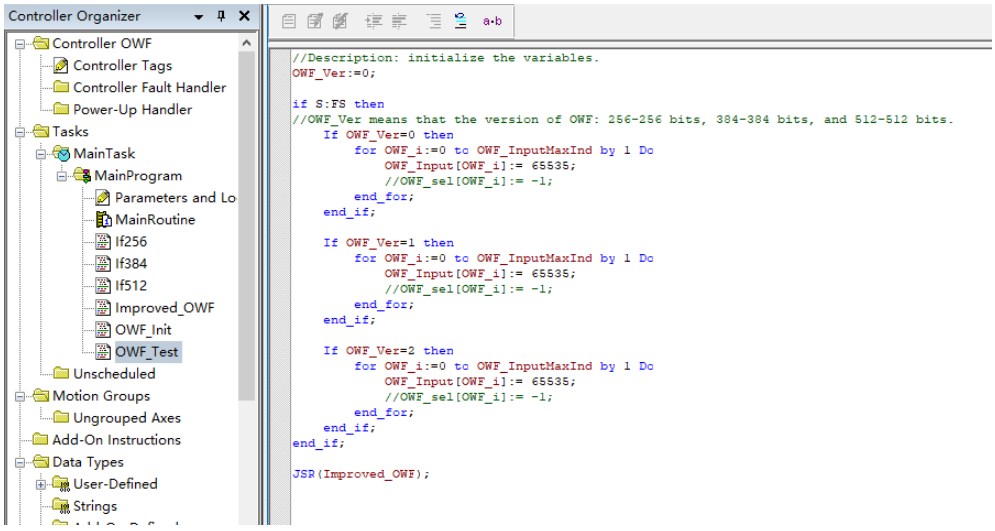

Figure 1. OWF Use-case

2. Universal one-way hash function (UOWHF)

(1) Open the project UOWHF.ACD;

(2) Init DINT variable **UOWHF_Ver** to be $j\ 2\ f0;1;2g$ to specify the version of UOWHF, where '0' denotes 256-bit input, '1' denotes 384-bit input, and '2' denotes 512-bit input;

(3) Assign the input values to tags $f\text{UOWHF\_Msg}[i]g_{i2[\text{NumDINT}]}$ where $\text{NumDINT}\ 2\ f8;12;16g$ are determined by **UOWHF_Ver**, e.g., 8*32=256 bits;

(4) Call **UOWHF**, via **JSR(UOWHF)**, to run the UOWHF Evaluation;

(5) Get the results in DINT ARRAY $f\text{UOWHF\_Result}[i]g_{i2[\text{NumDINT}]}$ where $\text{NumDINT}\ 2\ f5;7;9g$ corresponds to specific versions, respectively.

In Figure2, we show an example code of the above steps.

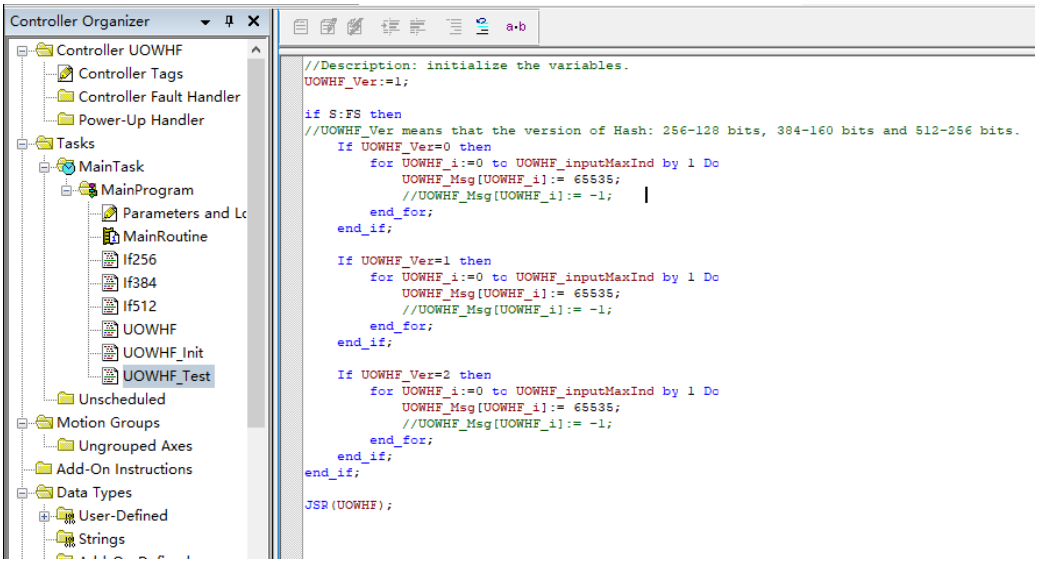

Figure 2. UOWHF Use-case

3. UOWHF with Variable Message Length

(1) Open the project UOWHF_Vary.ACD;

(2) Init DINT variable **UOWHF_Ver** to be $j\ 2\ f0;1;2g$ to specify the version of **UOWHF**, where '0' denotes 256-bit input, '1' denotes 384-bit input, and '2' denotes 512-bit input;

(3) Init DINT variable **UOWHF_Vary_MsgLen** to mean DINT number of message;

(4) Assign the input values to tags $f\text{UOWHF\_Vary\_Message}[i]g_{i2[\text{NumDINT}]}$ where $\text{NumDINT} = \text{UOWHF\_Vary\_MsgLen}$;

(5) Call **UOWHF_Vary**, via **JSR(UOWHF_Vary)**, to compute hash of Variable Message Length;

(6) Get the results in DINT ARRAY $f\text{UOWHF\_Result}[i]g_{i2[\text{NumDINT}]}$ where $\text{NumDINT}\ 2\ f5;7;9g$ corresponds to specific versions, respectively.

In Figure3, we show an example code of the above steps.

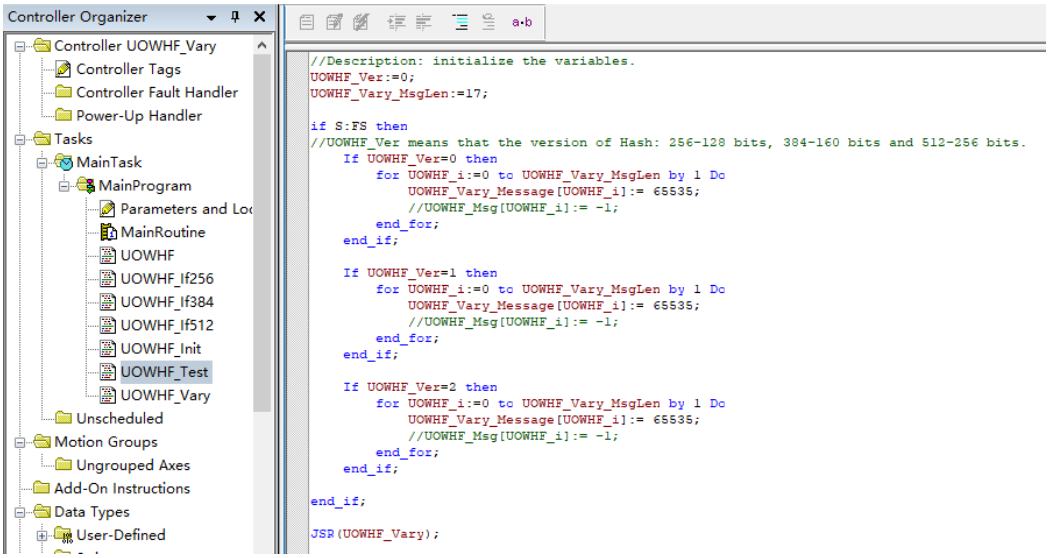

Figure 3. UOWHF_Vary Use-case

4. Message Authentication Code (CHASKEY)

(1) Open the project CHASKEY.ACD;

(2) Init DINT variable **Chaskey_MsgBitLen** to specify bit length of input message, and init DINT variable **Chaskey_round** to be $\{2, 8, 16\}$ to specify the round of permutation;

(3) Assign the input values to tags $\{Chaskey\_Message[i]\}_{i2[NumDINT]}$ where $NumDINT = Chaskey\_MsgBitLen/32$;

(4) Call **CHASKEY_128**, via **JSR(CHASKEY_128)**, to compute MAC;

(5) Get the results in DINT ARRAY$\{Chaskey_Result[i]\}_{i2[4]}$.

In Figure4, we show an example code of the above steps.

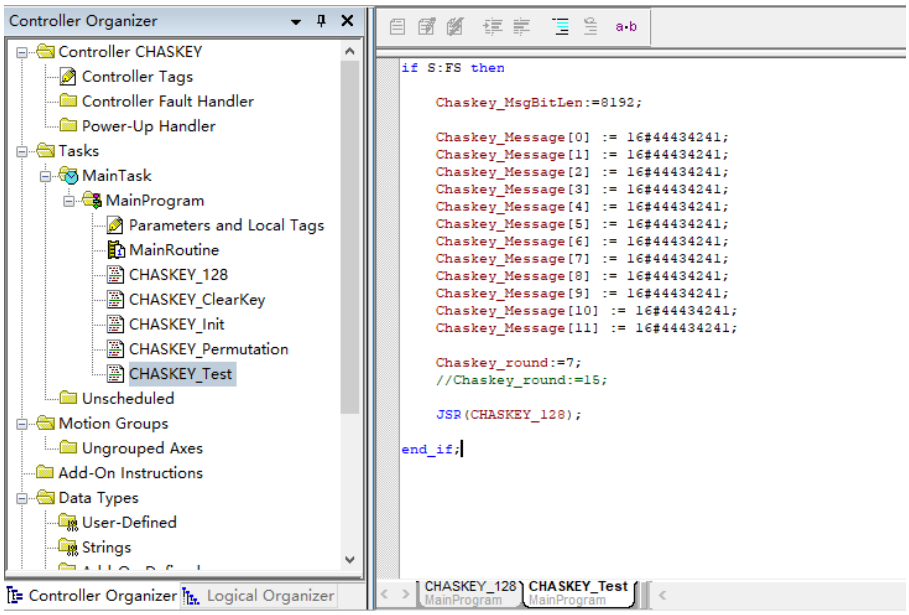

Figure 4. CHASKEY Use-case

5. Block Ciphers

5.1 PRESENT

(1) Open the project PRESENT_PreGenKey128(80).ACD;
(2) For PRESENT_Enc: Assign the input values to tags $\{PT\_Plaintext[i]\}_{i2[NumDINT]}$ where NumDINT = 2;
(3) Call **PRESENT_Enc**, via **JSR(PRESENT_Enc)**, to run the PRESENT Encryption;
(4) Get the results in DINT ARRAY $\{PT\_Ciphertext[i]\}_{i2[NumDINT]}$ where NumDINT = 2.

In Figure5, we show an example code of the above steps.

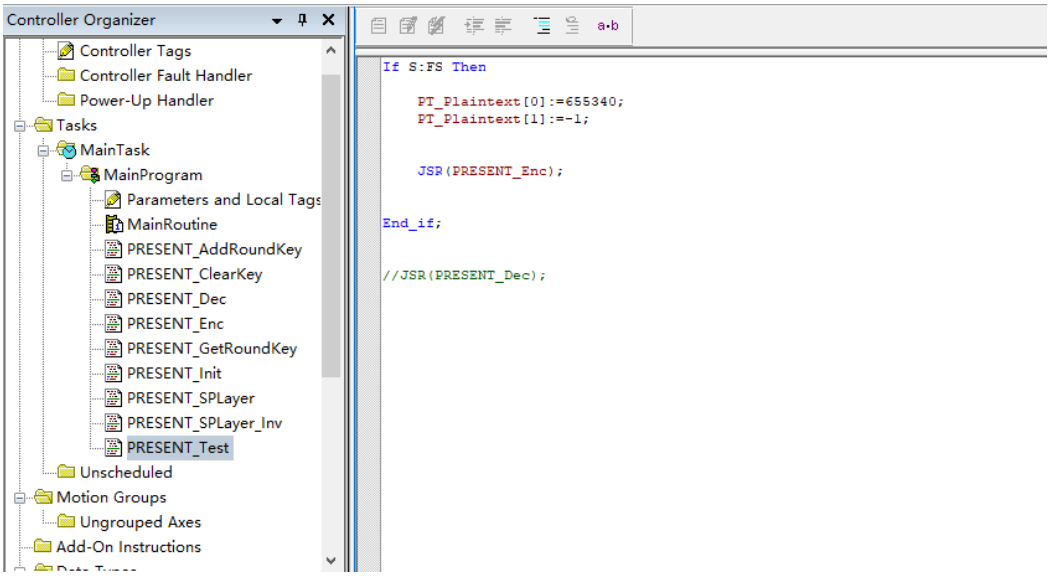

Figure 5. PRESENT Use-case

5.2 SPECK

(1) Open the project SPECK.ACD;
(2) For SPECK_Enc: Init DINT variable **Speck_Ver** to be $j \in \{0;1;2;3;4;5\}$ to specify the version of SPECK, where '0' denotes version 32/64, '1' denotes version 64/96, '2' denotes version 64/128, '3' denotes version 128/128, '4' denotes version 128/192 and '5' denotes version 128/256. The first number is bit-length of plaintext blocks and the second number is bit-length of key;
(3) Assign the input values to tags $\{Speck\_Plaintext[i;j]\}_{i2[2];j2[2]}$:
   a) If $j \in \{0;1;2\}$, assign the input values to tags $\{Speck\_Plaintext[i;0]\}_{i2[2]}$;
   b) If $j \in \{3;4;5\}$, assign the input values to tags $\{Speck\_Plaintext[i;j]\}_{i2[2];j2[2]}$;
(4) Call **SPECK_Enc**, via **JSR(SPECK_Enc)**, to run the SPECK Encryption;
(5) Get the results in DINT ARRAY $\{Speck\_Ciphertext[i;j]\}_{i2[2];j2[2]}$:
   a) If $j \in \{0;1;2\}$, assign the input values to tags $\{Speck\_Ciphertext[i;0]\}_{i2[2]}$;
   b) If $j \in \{3;4;5\}$, assign the input values to tags

f Speck_Ciphertext[i; j]g$_{i2[2];j2[2]}$;

*__Note__: the test-vector can be generated by the codes https://github.com/prashantbarca/simon-speck-sim/tree/master/Python

In Figure6, we show an example code of the above steps.

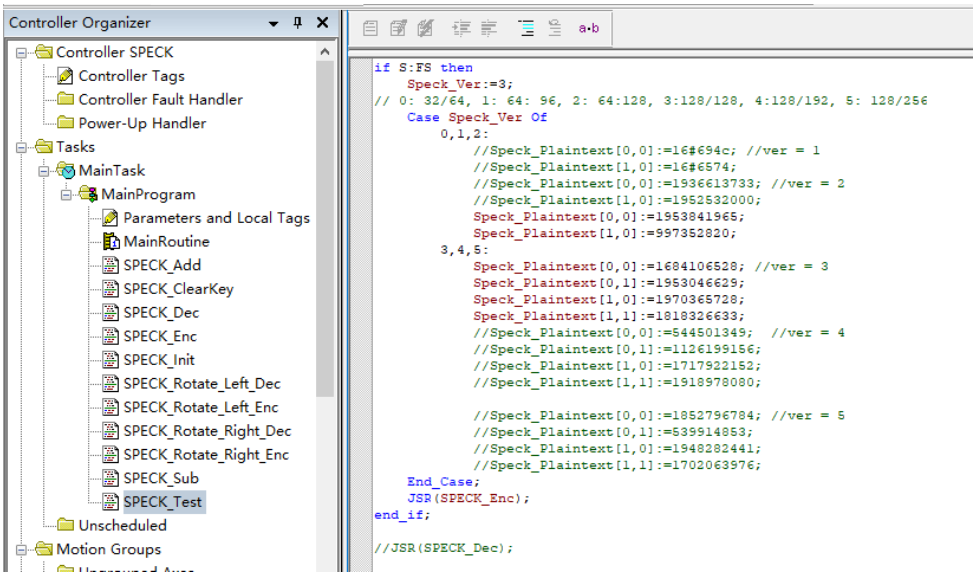

Figure 6. SPECK Use-case

5.3 SIMON

(1) Open the project SIMON.ACD;

(2) For SIMON_Enc: Init DINT variable **Simon_Ver** to be j 2 f0; 1; 2; 3; 4; 5g to specify the version of SIMON, where '0' denotes version 32/64, '1' denotes version 64/96, '2' denotes version 64/128, '3' denotes version 128/128, '4' denotes version 128/192 and '5' denotes version 128/256. The first number is bit-length of plaintext blocks and the second number is bit-length of key;

(3) Assign the input values to tags f Simon_Plaintext[i; j]g$_{i2[2];j2[2]}$:

   a) If j 2 f0; 1; 2g, assign the input values to tags f Simon_Plaintext[i; 0]g$_{i2[2]}$;

   b) If j 2 f3; 4; 5g, assign the input values to tags f Simon_Plaintext[i; j]g$_{i2[2];j2[2]}$;

(4) Call **SIMON_Enc**, via **JSR(SIMON_Enc)**, to run the SIMON Encryption;

(5) Get the results in DINT ARRAY f Simon_Ciphertext[i; j]g$_{i2[2];j2[2]}$:

   a) If j 2 f0; 1; 2g, assign the input values to tags f Simon_Ciphertext[i; 0]g$_{i2[2]}$;

   b) If j 2 f3; 4; 5g, assign the input values to tags f Simon_Ciphertext[i; j]g$_{i2[2];j2[2]}$;

In Figure7, we show an example code of the above steps.

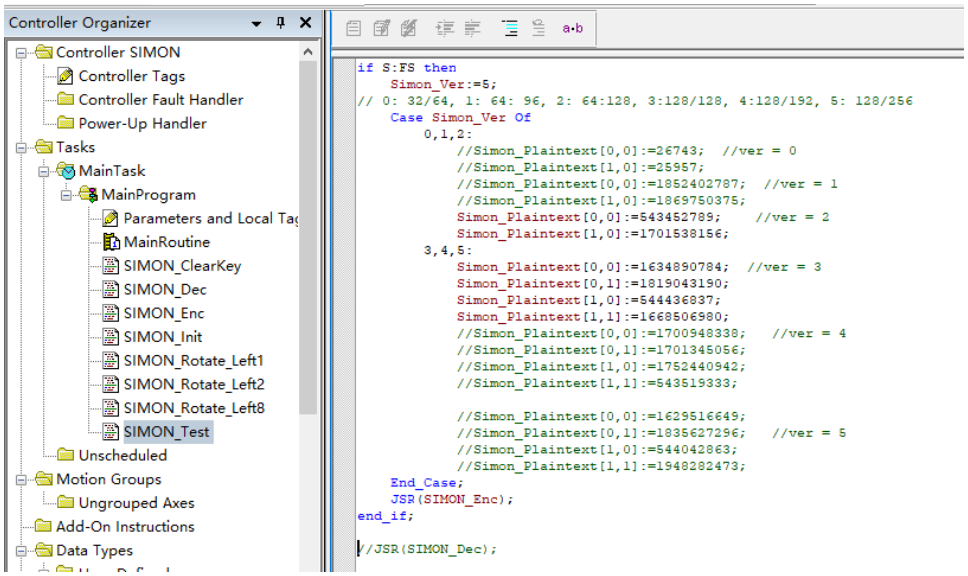

Figure 7. SIMON Use-case

6. **Pseudo-random function/generator (PRF/PRG)**

6.1 **PRF/PRG PRESENT**

(1) Open the project PRG_ PRESENT.ACD;

(2) Assign the input values to tags $\{PRG\_Counter[i]\}_{i2[NumDINT]}$ where NumDINT = 2, and PRG_Counter[0] defines the bit-length of output;

(3) Call **PRG_PRF_PRESENT**, via **JSR(PRG_PRF_PRESENT)**, to run the PRF/PRG Evaluation;

(4) Get the results in DINT ARRAY $\{PRG\_Result[i]\}_{i2[NumDINT]}$ where NumDINT = PRG_Counter[0] & 0xFFFF.

In Figure8, we show an example code of the above steps.

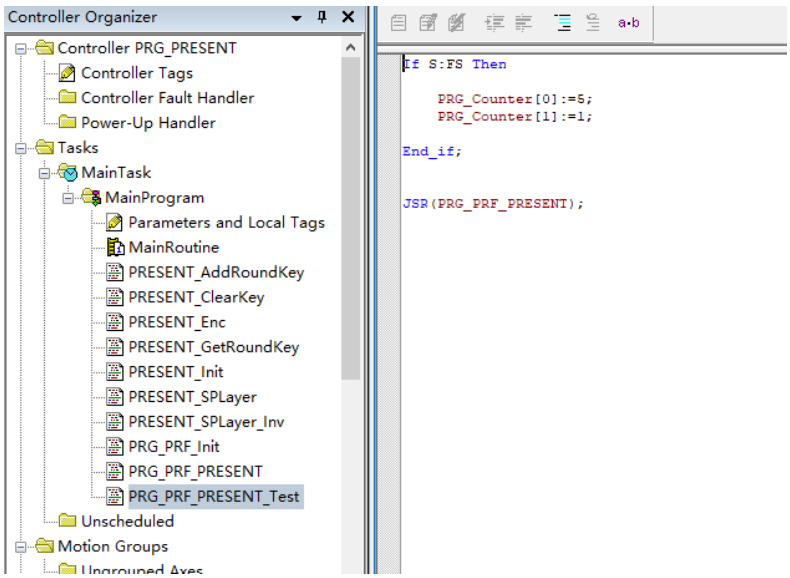

Figure 8. PRF/PRG PRESENT Use-case

6.2 **PRF/PRG SPECK**

(1) Open the project PRG_SPECK.ACD;

(2) Assign the input values to tags $\{PRG\_Counter[i]\}_{i2\,[NumDINT]}$ where NumDINT = 2, and PRG_Counter[0] defines the bit-length of output;

(3) Call **PRG_PRF_SPECK**, via **JSR(PRG_PRF_SPECK)**, to run the PRF/PRG Evaluation;

(4) Get the results in DINT ARRAY $\{PRG\_Result[i]\}_{i2\,[NumDINT]}$ where NumDINT = PRG_Counter[0] & 0xFFFF.

In Figure9, we show an example code of the above steps.

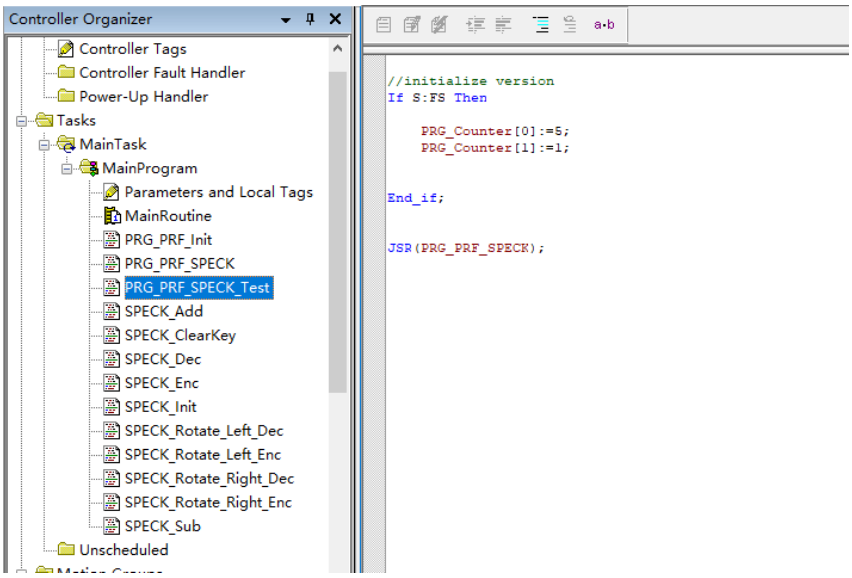

Figure 9. PRF/PRG SPECK Use-case

6.3 **PRF/PRG SIMON**

(1) Open the project PRG_ SIMON.ACD;

(2) Assign the input values to tags $\{PRG\_Counter[i]\}_{i2\,[NumDINT]}$ where NumDINT = 2, and PRG_Counter[0] defines the bit-length of output;

(3) Call **PRG_PRF_SIMON**, via **JSR(PRG_PRF_SIMON)**, to run the PRF/PRG Evaluation;

(4) Get the results in DINT ARRAY $\{PRG\_Result[i]\}_{i2\,[NumDINT]}$ where NumDINT = PRG_Counter[0] & 0xFFFF.

In Figure10, we show an example code of the above steps.

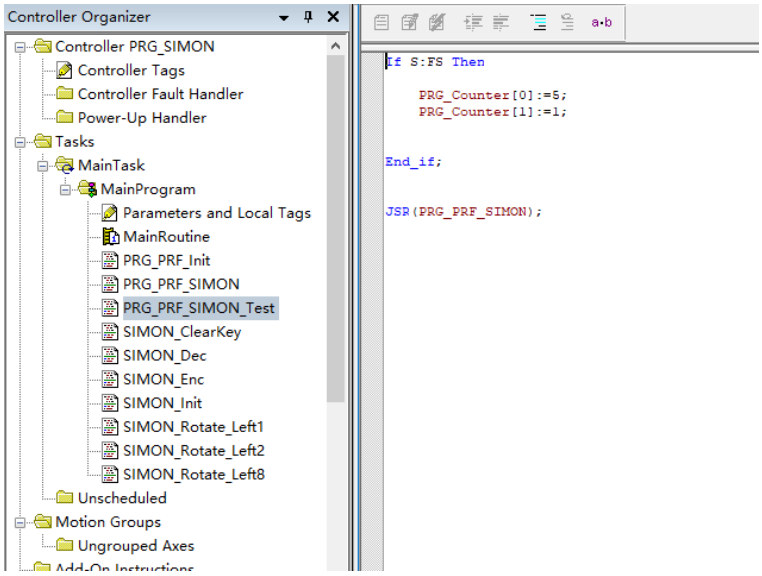

Figure 10. PRF/PRG SIMON Use-case

7. Collision-resistant hash function

7.1 PHOTON

(1) Open the project PHOTON.ACD;

(2) Init DINT variable **PHT_Ver** to be $j\ 2\ f0;1;2;3;4g$ to specify the version of PHOTON, where '0' denotes version 80/20/16, '1' denotes version 128/16/16, '2' denotes version 160/36/36, '3' denotes version 224/32/32, and '4' denotes version 256/32/32. The first number is bit-length of output hash, the second number is bit-length of input message block and the third number is bit-length of output block;

(3) Init DINT variable **PHT_MsgBitLen** to specify the bit-length of input message,like 2014, 2048. Assign the input values to tags $fPHT\_Message[i]g_{i2[NumDINT]}$ where $NumDINT = PHT\_MsgBitLen/8$;

(4) Call **PHOTON**, via **JSR(PHOTON),** to run the PHOTON Evaluation;

(5) Get the results in DINT ARRAY $fPHT_Digest[i]g_{i2[NumDINT]}$where $NumDINT = PHT\_DIGESTSIZE/8$.

In Figure11, we show an example code of the above steps.

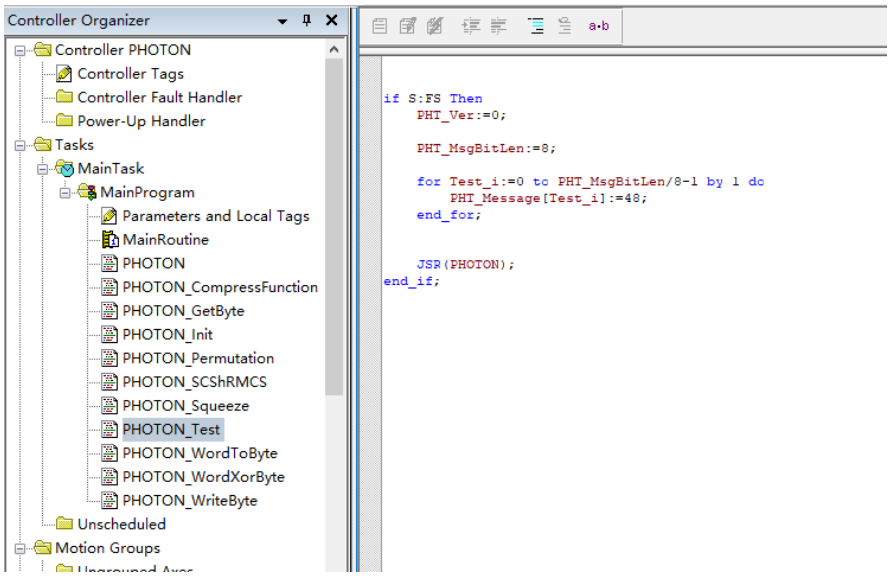

Figure 11. PHOTON Use-case

7.2 SPONGENT

(1) Open the project SPONGENT with Hard-coded PLayer 8817688.ACD, where '88' denotes bit-length of input message block, '176' denotes capacity, '88' denotes bit-length of output hash;

(2) Init DINT variable **Spon_Ver** to be $j\ 2\ f0;1;2;3;4g$ to specify the version of SPONGENT, where '0' denotes version 8817688, '1' denotes version 128/256/128, '2' denotes version 16016080, '3' denotes version 224/224/112, and '4' denotes version 256/256/128;

(3) Init DINT variable **Spon_MsgBitLen** to specify the bit-length of input message, like 2014, 2048. Assign the input values to tags $f\mathrm{Spon\_Message[i]}g_{i2[\mathrm{NumDINT}]}$ where $\mathrm{NumDINT} = \mathrm{PHT\_MsgBitLen}/8$;

(4) Call **SPONGENT**, via **JSR(SPONGENT),** to run the SPONGENT Evaluation;

(5) Get the results in DINT ARRAY $f\mathrm{Spon_Digest[i]}g_{i2[\mathrm{NumDINT}]}$where $\mathrm{NumDINT} = \mathrm{Spon\_DIGESTSIZE}/8$.

In Figure12, we show an example code of the above steps.

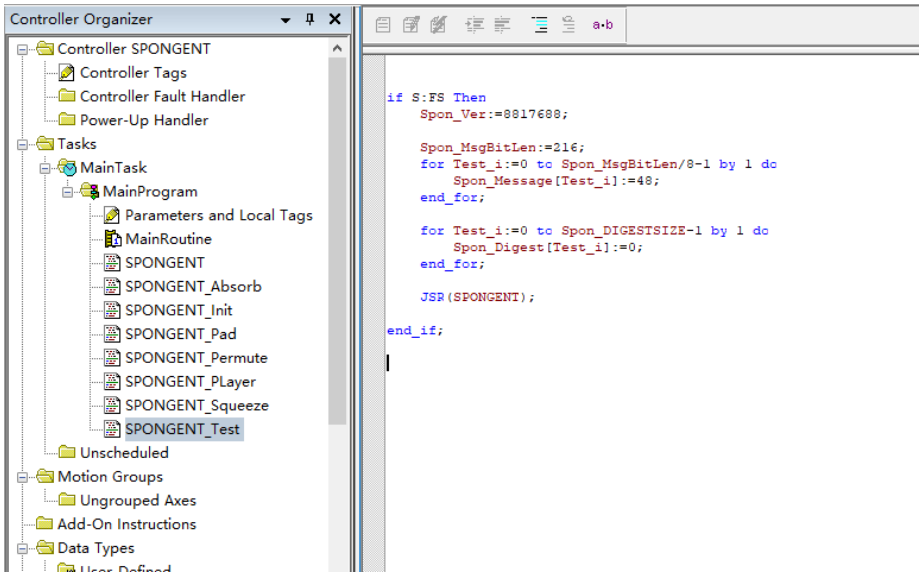

Figure 12. SPONGENT Use-case

8. **Proof of Aliveness (POA)**
   (1) Open the project POA.ACD;
   (2) Call POA_ProofGeneration, via JSR(**POA_ProofGeneration**)

Then, the PoA would run automatically with default sub-chain length **POA_Pf_Index**=64, and a proof generation time interval **POA_Pf_Interval**=1000. The routine **POA_Replenishment** for proof replenishment would be invoked by **POA_ProofGeneration** automatically when the proofs are used out.

*__Note__:Users who want to customize the POA instances, could open **POA_PF_Init** routine and change the above two tags accordingly**.**

In Figure13, we show an example code of the above steps.

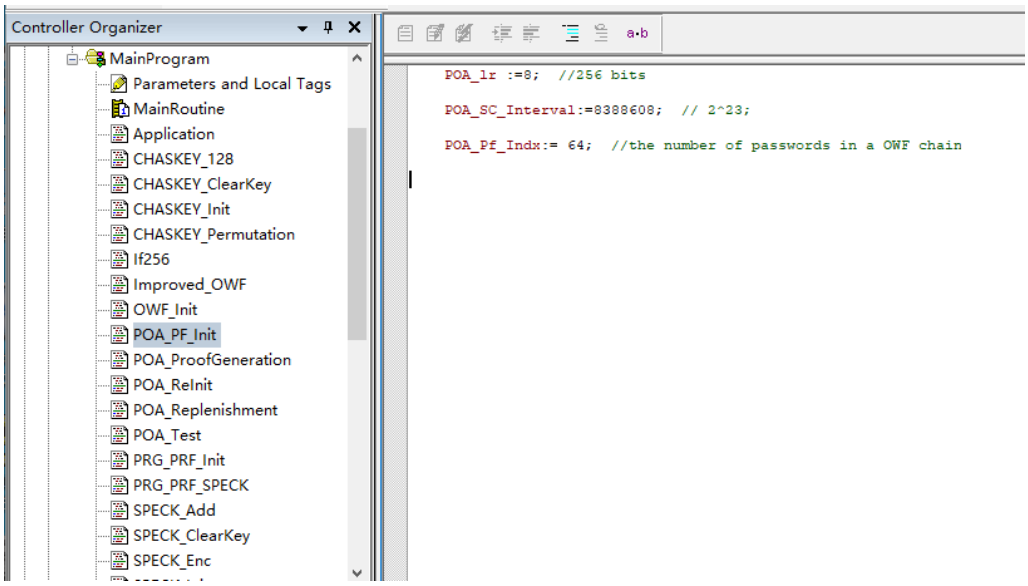

Figure 13. POA Use-case

9. Big-integer operations

9.1 Multiple-precision addition

(1) Open the project Big-integer.ACD;
(2) Init DINT variable **Big_Ver** to be $j \ 2 \ f0; 1; 2g$ to specify the version of base, where '0' denotes base = 10, '1' denotes base = 2^{15}, and '2' denotes base = 2^{30};
(3) Init DINT variable **Add_Digits** regarding the digit numbers;
(4) Assign the input values to tags $f\text{Add\_input\_x}[i]g_{i2\,[\text{Add\_Digits}]}$ and $f\text{Add\_input\_y}[i]g_{i2\,[\text{Add\_Digits}]}$;
(5) Call **BIGINTEGER_Add**, via **JSR(BIGINTEGER_Add),** to compute the sum;
(6) Get the results in DINT ARRAY $f\text{Add_Result}[i]g_{i2\,[\text{NumDINT}]}$where NumDINT = Add_Digits + 1.

9.2 Multiple-precision subtraction

(1) Open the project Big-integer.ACD;
(2) Init DINT variable **Big_Ver** to be $j \ 2 \ f0; 1; 2g$ to specify the version of base, where '0' denotes base = 10, '1' denotes base = 2^{15}, and '2' denotes base = 2^{30};
(3) Init DINT variable **Sub_Digits** regarding the digit numbers;
(4) Assign the input values to tags $f\text{Sub\_input\_x}[i]g_{i2\,[\text{Sub\_Digits}]}$ and $f\text{Sub\_input\_y}[i]g_{i2\,[\text{Sub\_Digits}]}$;
(5) Call **BIGINTEGER_Sub**, via **JSR(BIGINTEGER_Sub),** to compute the difference;
(6) Get the results in DINT ARRAY$f\text{Sub_Result}[i]g_{i2\,[\text{Sub_Digits}]}$.

9.3 Multiple-precision multiplication

(1) Open the project Big-integer.ACD;
(2) Init DINT variable **Big_Ver** to be $j = 1$ to specify the version of base, where '1' denotes base = 2^{15};
(3) Init DINT variable **Mul_Digits** regarding the digit numbers;
(4) Assign the input values to tags $f\text{Mul\_input\_x}[i]g_{i2\,[\text{Mul\_Digits}]}$ and $f\text{Mul\_input\_y}[i]g_{i2\,[\text{Mul\_Digits}]}$;
(5) Call **BIGINTEGER_Mul**, via **JSR(BIGINTEGER_Mul),** to compute the product;
(6) Get the results in DINT ARRAY $f\text{Mul_Result}[i]g_{i2\,[\text{NumDINT}]}$where NumDINT = 2 ¤ Mul_Digits.

9.4 Multiple-precision division

(1) Open the project Big-integer.ACD;
(2) Init DINT variable **Big_Ver** to be $j = 1$ to specify the version of base, where '1' denotes base = 2^{15};
(3) Init DINT variable **Div_x_Digits** and **Div_y_Digits** regarding the digit numbers;
(4) Assign the input values to tags $f\text{Div\_input\_x}[i]g_{i2\,[\text{Div\_x\_Digits}]}$ and $f\text{Div\_input\_y}[i]g_{i2\,[\text{Div\_y\_Digits}]}$ where Div_x_Digits = 2 ¤ Div_y_Digits;
(5) Call **BIGINTEGER_Div**, via **JSR(BIGINTEGER_Div),** to compute the quotient and remainder;

(6) Get the results in DINT ARRAY f Div_quotient[i]g$_{i2\,[NumDINT]}$where NumDINT = Div_x_Digits ¡ Div_y_Digitsand f Div_remainder[i]g$_{i2\,[Div_y_Digits]}$.

9.5 Multiple-precision additive multiplication

(1) Open the project Big-integer.ACD;
(2) Init DINT variable **Big_Ver** to be j = 2 to specify the version of base, where '2' denotes base = 2^{30}, AddMul_Digits = 9;
(3) Assign the input values to tags f AddMul_base[i]g$_{i2\,[AddMul_Digits-1]}$, f AddMul_input_g[i]g$_{i2\,[AddMul_Digits-1]}$ and f AddMul_exp[i]g$_{i2\,[AddMul_Digits-1]}$;
(4) Assign the input values to tags f Mul_input_x[i]g$_{i2\,[AddMul_Digits-2]}$ and f Mul_input_y[i]g$_{i2\,[AddMul_Digits-2]}$;
(5) Call **BIGINTEGER_AMul**, via **JSR(BIGINTEGER_AMul),** to compute the product;
(6) Get the results in DINT ARRAYf AddMul_Result[i]g$_{i2\,[AddMul_Digits]}$.

In Figure14, we show an example code of the above steps.

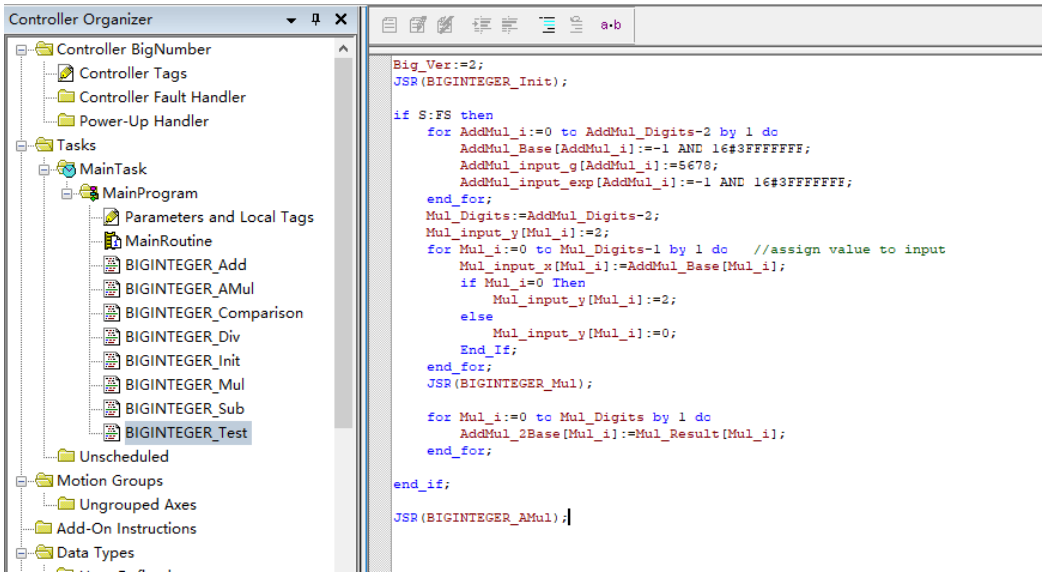

Figure 14. Big-integer Use-case