Message Based Processing

Message based processing permits the processing of a single message through a single call to the API. Considering this from the existing API, a message consists of the data processed starting with the first call to one of the C_*Init functions, continuing through any C_*Update calls and ending with the C_*Final call, (or for the short form using encryption as an example from the C_EncryptInit through C_Encrypt.). In many protocol cases, the same key and mechanism are used repeatedly to process many messages. By using a message-based approach, this can limit the number of round-trips between an HSM and a client program and improve performance and latency.

This message based API encapsulates the common elements of per-message data in a session object called an 'association'. The handle for that association object is passed as part of the per-message processing and is used initialize the cryptographic state of the HSM session prior to processing the message data.

In its simplest form, message processing can be thought of as a macro which translates to calls to C_*Init/C_*Update/C_*Final, where the initialization data for C_*Init such as keys and mechanisms is taken from the association data. In the simplest case, using the API below, a call to C_ProcessMessage for encryption would translate to the equivalent of a call to C_EncryptInit followed by a call to C_Encrypt.

In addition to the key and cryptographic mechanism inputs, the message API allows the specification of an IV generator. The IV generator is used to supply a per-message IV, and the IV used is returned as part of the message processing calls.

Lastly, the API below is designed to handle processing of encryption and decryption (including AEAD based mechanisms), and signing and verification. Because of that it includes arguments that may not be used in all cases. Specifically, non-AEAD mechanism will not use all arguments.

PKCS11 2.40 limited the number of in-progress cryptographic operations per session to one of encryption/decryption and one of hash/sign/verify\(^1\). If an HSM implements the message based API, the HSM MAY relax that restriction to permit one active encrypt/decrypt context and one active hash/sign/verify per association rather than per session. For message-based processing, cryptographic contexts are active from a call to C_ProcessMessage with ulPhase equal to PHASE_INIT until a call with ulPhase equal to PHASE_FINAL or during a call with ulPhase equal to one of the PHASE_SINGLE_* values. For backwards compatibility, a program should assume the previous limits unless it is configured otherwise. [Design note - we really need a more general way of providing per-token configuration information than C_GetTokenInfo]

\(^1\) AEAD mechanisms are logically the combination of an encrypt/decrypt context with a sign/verify context, but may not actually be implemented that way by a token.
PKCS11 - Message Processing - V1.1 - 5 May 2014

Message Processing API Calls
* C_NewAssociation

```c
CK_DEFINE_FUNCTION (CK_RV, C_NewAssociation) {
    CK_SESSION_HANDLE    hSession,
    CK_OBJECT_HANDLE_PTR phAssociationHandle,
    CK_MECHANISM_PTR     pMechanism,
    CK_OBJECT_HANDLE     hKey,
    CK_MECHANISM_PTR     pIvGeneratorMechanism,
    CK_BBOOL             bReceiver
};
```

**C_NewAssociation** creates a container of objects used for message-based encryption, decryption, signing, and verification. *hSession* is the session handle; *phAssociationHandle* is the location used to store the returned handle of the object used to wrap the collection; *pMechanism* is the encryption or signature (cryptographic) mechanism that will be used within the association; *hKey* is the key used within the association; *pIvGeneratorMechanism* is the optional mechanism used to generate per-message initialization vectors or NULL_PTR if no mechanism is needed (for example because IVs are being generated externally); and *bReceiver* is CK_FALSE if this association will be used by a sender (doing encryption or signing) or CK_TRUE if being used by a receiver (doing decryption or verification).

Certain types of mechanisms may benefit from one-time cryptographic setup or initializations specific to the key or mechanism that will be common across all messages (e.g. key schedule expansion for AES). It is an implementation choice whether to do this as part of the call to **C_NewAssociation** or to defer it until the beginning of the first call to **C_ProcessMessage** using this association.

Certain cryptographic mechanisms may restrict the types of IV generators that are used with them. If the specified IV generator is either prohibited or incompatible with the cryptographic mechanism, then the call will return a CKR_XXX error.

[Design discussion - it's unclear that a copy of the association object should be permitted as it may contain IV generator state and duplicating that and reusing it could be bad. If a copy of the association is permitted then the following language applies: [The association object returned by a call to **C_NewAssociation** is an opaque object with session scope. It has no readable or writable attributes, but may be cloned through the use of **C_CopyObject**, however any attempt to add or change attributes during the copy will result in a failure CKR_XXX.] Otherwise if no copy is permitted then this language applies: [The association object returned by a call to C_NewAssociation is an opaque object with session scope. It has no readable or writable attributes and may not be copied via **C_CopyObject** or changed via **C_SetAttributeValue**.]]

[Design discussion - the above uses a discrete explicit association object to contain association state as opposed to an implicit association state within the session state. This approach permits more than one}
association to be used per session but if this isn't necessary, then the relevant arguments above and below can be removed, and language changed.]

* C_ProcessMessage

```c
CK_DEFINE_FUNCTION (CK_RV, C_ProcessMessage) (  
    CK_SESSION_HANDLE  hSession,  
    CK_OBJECT_HANDLE  hAssociationHandle,  
    CK_BYTE_PTR  pInData,  
    CK_ULONG  ulInDataLen,  
    CK_BYTE_PTR  pAssociatedData,  
    CK_ULONG  ulAssociatedDataLen,  
    CK_BYTE_PTR  pOutData,  
    CK_ULONG_PTR  pulOutDataLen,  
    CK_BYTE_PTR  pIntegrityTag,  
    CK_ULONG  pulIntegrityTagLen,  
    CK_BYTE_PTR  pIv,  
    CK_ULONG  ulIvLen,  
    CK_ULONG  ulPhase  
);  

#define PHASE_SINGLE 0  
#define PHASE_SINGLE_PREFIX 10  
#define PHASE_SINGLE_POSTFIX 11  
#define PHASE_INITIAL 1  
#define PHASE_UPDATE 2  
#define PHASE_FINAL 3
```

[Design discussion: C_ProcessMessage takes the place of up to 16 new API calls (encrypt/decrypt/sign/verify x processone/processfirst/processupdate/processfinal). While the definition has quite a lot of moving parts, it's pretty easy to make macros from this single call that behave as if there were 16 implemented functions.

**C_ProcessMessage** is used to process sent or received messages within a defined association. **C_ProcessMessage** can process either a complete message in one pass or be used to process a message in segments depending on the value for ulPhase. hSession is the session handle; hAssociationHandle is the association handle returned by **C_NewAssociation**; pInData is the data to be processed for an encryption/decryption association, either the plaintext or cipher text depending on direction or NULL_PTR if the association mechanism is a signature only mechanism; ulInDataLen is the length of the plaintext or ciphertext; pAssociatedData is the additional data for an AEAD mechanism or the data to be signed or verified for signature only mechanism; ulAssociatedDataLength is the length of the associated data for an AEAD association or to-be-signed data for a signature only association; pOutData points to the location that receives the encrypted or decrypted data; pulOutDataLen points to the location that contains the length of the output data; pIntegrityTag points to the location that receives the signature
or integrity tag of the data; pullIntegrityTagLen contains a pointer to the location of the length of the integrity tag; plv points to the location that contains or receives a per-message initialization vector or NULL_PTR if one is not required or output; ulvLength contains the length of the lv input data; ulPhase is the init/update/final or single part signing phase - see below.

The actual processing to be done for each call can be inferred based on the association's mechanism type and whether this association is a sender or receiver. For example, if the association mechanism is an encryption or AEAD mechanism, and bReceiver is CK_TRUE, then this is a decryption operation. Or if the association mechanism is a signature mechanism and bReceiver is CK_FALSE then this is a signature mechanism.

If ulPhase is either PHASE_SINGLE, PHASE_SINGLE_PREFIX or PHASE_SINGLE_POSTFIX then this call to C_ProcessMessage processes a complete message in a single pass. Otherwise, this behaves similar to the C_*Init/C_*Update/C_*Final conventions defined elsewhere. By convention, unless specified otherwise for the specific mechanism, an AEAD mechanism processes the associated data in its entirety prior to performing encryption or decryption. If permitted by the mechanism, PHASE_SINGLE_PREFIX or PHASE_SINGLE_SUFFIX may be used to override this ordering and process the associated data either before or after encryption or decryption. If ulPhase is PHASE_INIT, then the associated data, if any, is processed before encryption or decryption. If ulPhase is PHASE_FINAL, then the associated data, if any, is processed after encryption or decryption. If ulPhase is PHASE_UPDATE, then the associated data is ignored. Two calls to C_ProcessMessage with the first call where ulPhase is PHASE_INIT and the last call where ulPhase is PHASE_FINAL may be used, if permitted by the mechanism, to provide associated data before and after the plain or cipher text.

Figure 1 - Permitted Calling Sequences For Message-Based Processing

An out of sequence call such as calling C_ProcessMessage with ulPhase of PHASE_SINGLE (any variety) after making a C_ProcessMessage/PHASE_INIT call and before making a C_ProcessMessage/PHASE_FINAL call terminates the processing of the current message and returns an error CKR XXX.
If the mechanism specified by the association is an encryption or AEAD mechanism, then *pInData* points either to the ciphertext to be decrypted or plaintext to be encrypted depending on the direction of the association. A specific AEAD mechanism MAY restrict release of decrypted plaintext until verification is successful, but this API places no such limits.

*pOutData* and *pulOutDataLen* are used to contain the output of an encryption or decryption. The processing for these fields follows the conventions of section 11.2 [REVISE SECTION NUMBER].

If the mechanism specified by the association is a signature only mechanism, the data to be signed or verified is pointed to by *pAssociatedData*. *pInData* is ignored and should be set to NULL_PTR.

For an AEAD or signature only mechanism, the input or output signature/integrity tag SHOULD be placed in the location pointed to by *pIntegrityTag*. A specific AEAD mechanism may override this and provide the integrity tag/signature as part of the input or output ciphertext. The API supports either approach. For output of signatures (i.e. when *bReceiver* is CK_FALSE), the processing of *pulIntegrityTagLen* follows the conventions of section 11.2[REVISE]. For input of signatures where the signature is not part of the input ciphertext, *pulIntegrityTagLen* must point to a location that contains the length of the signature. 

[Design discussion: It turns out not to be the case that the output length of a signature is always known - cf ECDSA which uses an ASN1 encoding for an output signature and which can vary by a couple of bytes either way on a per signature basis]

The *plv* value interacts with the association *plvGeneratorMechanism* value in the following ways:

1. If *plvGeneratorMechanism* is NULL_PTR and the association mechanism requires an IV, then *plv* must not be NULL_PTR and points to a location that contains the message's IV regardless of whether the association is a send or receive association. CKR_XXX is returned if both *plvGeneratorMechanism* and *plv* are NULL_PTR.
2. The association mechanism or operational mode of the HSM MAY require that *plvGeneratorMechanism* be set (e.g. for a FIPS implementation of AES-CCM or AES-GCM on the sending side), but that is not an API requirement. CKR_XXX is returned if an IV generator is required, but not specified.
3. If *plvGeneratorMechanism* is set, then after a successful call to **C_ProcessMessage** where *ulPhase* is either PHASE_SINGLE* or PHASE_INIT, AND *plv* is non-null, then *plv* will contain the initialization vector used to encrypt this message. The IV generator state is updated in this event. Note: while it is unlikely that an IV generator will be used on the received side for to-be-defined mechanisms, the API does not prohibit this.
4. Certain IV generator mechanisms MAY require an input value on a per-message basis. That value will be placed at the location specified by *plv*. After a successful call to **C_ProcessMessage** as in (3) above then the location specified by *plv* will contain the generated IV. In this case, *ullvLen* contains the length of the input parameter and does not necessarily contain the length of the output IV. Sufficient space should be provided at *plv* to contain the full length of the output IV without reference to this value.
5. [Design Discussion: Note that \( ul\text{IVLen} \) is an input-only value. It is provided mainly to ease network marshalling/unmarshalling, but for most mechanisms the length of the IV should be fixed. However, there are some possible approaches to IV generators where the input parameter length (4 above) is not the same as the output IV length. I’m thinking about how CBC IVs are generated for example. Its possible this parameter is completely redundant with the association information and could be removed. Alternately, it could make sense to separate IV input and IV output fields. The language for this field is still subject to some tweaking].

\* C\_CloseAssociation

```c
CK_DEFINE_FUNCTION (CK_RV, C_CloseAssociation) (
    CK_SESSION_HANDLE  hSession,
    CK_OBJECT_HANDLE  hAssociationHandle
);
```

\textbf{C\_CloseAssociation} terminates any cryptographic operations in progress for the association, and releases any storage and processing objects (e.g. IV generator) contained within the association.

\textbf{IV Processing for Legacy Mechanisms}
Existing encryption mechanisms that use an IV may use the message processing API as follows:

1. The IV field specified in the mechanism parameters of the cryptographic mechanism is ignored.
2. An IV generator may be specified as an argument to \textbf{C\_NewAssociation}, but that is not required.
3. If an IV generator is not specified, then the IV of appropriate length \textbf{MUST} be provided at \( p\text{lv} \)
   and is used in place of the IV that would normally be supplied as part of the mechanism parameters.
4. If an IV is not supplied (\textit{e.g.} \( p\text{lv} \) is NULL\_PTR in an initial call to \textbf{C\_ProcessMessage} and there is no IV generator), the call will fail with a CKR\_XXX.

\textbf{Other Message Processing Functions}
It is trivial to define macros for all of the various permutations of the usages of \textbf{C\_ProcessMessage} to accomplish message full and multi-part signing, verifying, encrypting and decrypting. For example, a message function that does a single part signing would have the following definition:

```c
#define C\_SignMessage ( \`
    hSession, hAssociationHandle, \`
    pAssociatedData, ulAssociatedDataLength, \`
    pIntegrityTag, pulIntegrityTagLength) \`
C\_ProcessMessage( \`
    hSession, hAssociationHandle, NULL\_PTR, 0, \`
    pAssociatedData, ulAssociatedDataLength, \`
    NULL\_PTR, NULL\_PTR, \`
    pIntegrityTag, pulIntegrityTagLength, \`
    NULL\_PTR, 0, PHASE\_SINGLE);```
Initialization Vector Generators

IV generators are defined for use as part of the message-based processing constructs. They are specified as part of the call to `C_NewAssociation` and provide a way to automatically supply an IV for each message processed within the association. If the association contains an IV generator, every initial call to `C_ProcessMessage` (i.e. where `ulPhase` is either one of the PHASE_SINGLE values or is PHASE_INITIAL) provides an IV for that specific message, and updates the IV generator state (if any). If `plv` is non-null, then the provided IV will be output to the location specified.

Some IV generators MAY take an input value. That value is also placed at the location specified by `plv`, and the value of `ulvLength` indicates the length of the input or is '0' if there is no input. Note that the length of the output IV is specific to the association mechanism and need not be specified in the call to `C_ProcessMessage`. [Design Discussion: I've seen hints of people wanting to use hashes of sequence numbers in various proprietary protocols - so the input would be a sequence number, the IV would be the left truncate of that hash for example]

* **CKM_IVGEN_CCM_COUNTER**

```c
typedef struct CK_IVGEN_CCM_COUNTER_PARAMS {
    CK_BYTE_PTR      pNonce,
    CKULONG          ulNonceLength,
    CKULONG          ulFixedFieldLength
} CK_IVGEN_CCM_COUNTER_PARAMS;
```

This mechanism generates the per-message nonce for CCM. It was designed to work for a counter block formatted as per Appendix A.3 of SP800-38C, but may be used for other counter formats. It takes a parameter of type `CK_IVGEN_CCM_COUNTER_PARAMS`. `ulNonceLength` is the total length of the Appendix A nonce 'N' in bytes and must be between 7 and 13 inclusive; `pNonce` points to the location that contains the fixed part of the nonce or is NULL_PTR; `ulFixedFieldLength` is the length of the fixed part of the nonce in bytes and must be less than `ulNonceLength`.

If `pNonce` is NULL_PTR, then a random value will be set for the fixed part of the nonce, and the initial value for the variable part of the nonce will be the big endian representation of '1'. I.e. if the length of the variable part is 4, then the value of the variable part will be [0x00 0x00 0x00 0x01].

A new nonce is generated for each message processed within a send association. The value of the nonce is the concatenation of the fixed part of the nonce (either specified or generated) and the variable part of the nonce. The next nonce is formed by treating the variable part of the nonce as a big-endian integer and incrementing it by 1. Note that if `pNonce` is not null, then the rightmost `ulFixedFieldLength` bytes are the initial value of that counter.

[Design note - this was specified the way it was so that it wasn't tied to the non-normative Appendix A of the CCM standard and could be used with other format functions]
**CKM_IVGEN_GCM_DETERMINISTIC**

```c
typedef struct CK_IVGEN_GCM_DETERMINISTIC_PARAMS {
    CK_BYTE_PTR pFirstIv,  
    CK_ULONG ulIvLength,  
    CK_ULONG ulFixedFieldLength
} CK_IVGEN_GCM_DETERMINISTIC_PARAMS;
```

This is the NIST SP800-38D section 8.2.1 IV construct. This mechanism generates an IV of any length for use with Galois Counter Mode (GCM) encryption and takes a parameter of CK_IVGEN_GCM_DETERMINISTIC_PARAMS. `pFirstIv` points to the location that contains the IV of the first message to be processed; `ulIvLength` contains the length of all IVs to be used with the association, including the first IV; `ulFixedFieldLength` contains the length of the fixed part of the IV field and must be less than `ulIvLength`.

The left-most `ulFixedFieldLength` bytes in the IV are constant across all messages. The right-most `ulIvLength-ulFixedFieldLength` bytes are treated as a big-endian integer and that integer is incremented after each call to `C_ProcessMessage`.

[Design note - this is pretty similar to the definition of CKM_IVGEN_CCM_COUNTER and its possible they could be combined and renamed]

**CKM_IVGEN_RANDOM**

```c
typedef struct CK_IVGEN_RANDOM_PARAMS {
    CK_ULONG ulRandomFieldLength,  
    CK_BYTE_PTR pFreeField,  
    CK_ULONG ulFreeFieldLength,  
    CK_MECHANISM_TYPE randomMechanism
} CK_IVGEN_RANDOM_PARAMS;
```

This is the NIST SP800-38D section 8.2.2, first option construct. It generates an IV of any length and takes a parameter of CK_IVGEN_RANDOM_PARAMS. `ulRandomFieldLength` is the length of the random part of the IV; `pFreeField` points to the location of the fixed part of the IV - may be NULL_PTR if no fixed portion; `ulFreeFieldLength` contains the length of the fixed portion - should be 0 if `pFreeField` is NULL_PTR; and `randomMechanism` is the mechanism to use to generate the random field for each per-message IV.

The IV is the concatenation of the random field and the free field. For each message, a new random value is generated and concatenated to the supplied free field to form the IV.

This may be used with an mechanism that requires a random IV. For use with GCM, `ulRandomFieldLength` must be >= 12.
**CKM_IVGEN_PSEUDO_RANDOM**

```c
typedef struct CK_IVGEN_PSEUDORAND_PARAMS {
    CK_BYTE_PTR  pSeed,
    CK_ULONG  ulSeedLength,
    CK_MECHANISM_TYPE randomMechanism
} CK_IVGEN_PSEUDORAND_PARAMS;
```

This mechanism generates a pseudorandom IV of any length. It takes a parameter of `CK_IVGEN_PSEUDORAND_PARAMS`. `pSeed` points to the location containing the seeding information; `ulSeedLength` is the length of the provided seed material; and `randomMechanism` is the pseudorandom mechanism type. Given the same seed and same `randomMechanism` type, it always produces the same sequence of IVs.

**Message Based Processing Examples**

**AES-CCM - Sender**

```c
CK_BYTE[23] message1;
CK_BYTE[10] aad1;

CK_BYTE[100] outbuf;
CK_BYTE[8] integrityTag;
CK_ULONG outbufLen;
CK_ULONG integrityTagLen;

CK_BYTE[16] iv; // fixed length IV for this mode.

CK_IVGEN_CCM_COUNTER_PARAMS ccmIvParams;
CK_MECHANISM ivGenMech;
CK_CCM_PARAMS ccmParams;
CK_MECHANISM encryptMech;
CK_HANDLE hAssociation;
CK_RV result;

ccmParams.pNonce = NULL_PTR;
ccmParams.ulNonceLength = 13;
ccmParams.ulFixedFieldLength = 9; // 32 bit counter
```
ivGenMech.mechanism = CKM_IVGEN_CCM_COUNTER;
ivGenMech.pParameter = &ccmIvParams;
ivGenMech.ulParameterSize = sizeof(CK_IVGEN_CCM_COUNTER_PARAMS);

ccmParams.ulMacLen = 8; // only this needs to be set, all others
  // ignored for message based as they're
  // available elsewhere
encryptMech.mechanism = CKM_AES_CCM;
encryptMech.pParameter = &ccmParams;
encryptMech.ulParameterSize = sizeof(CK_CCM_PARAMS);

result = C_NewAssociation (hSession, &hAssociation,
  &encryptMech, encryptKey, &ivGenMech, CK_FALSE); // sender

outbufLen = 100;
integrityTagLen = 8;

// For each sent message

result = C_ProcessMessage (hSession, hAssociation,
  message1, 23, aad1, 10, outbuf, &outbufLen,
  integrityTag, &integrityTagLen, iv, 0, // no input iv data
  PHASE_SINGLE_POSTFIX);

// marshal protocol data

// send message

// end for loop.

// Close the association

result = C_CloseAssociation (hSession, hAssociation);
HMAC-SHA256 - Receiver

CK_BYTE[100] associatedData1;
CK_ULONG associatedDataLen1 = 100;
CK_BYTE[100] associatedData2;
CK_ULONG associatedDataLen2 = 100;

CK_BYTE[8] signature;
CK_ULONG sigLen = 8;
CK_HANDLE hSession;
CK_HANDLE hAssociation;

CK_HANDLE sigKey;

CK_MAC_GENERAL_PARMS macParms = 8; // 8 signature byte tag.

CK_MECHANISM sigMechanism;
sigMechanism.mechanism = CKM_SHA256_HMAC_GENERAL;
sigMechanism.pParameter = &macParms;
sigMechanism.ulParameterLen = sizeof(macParms);

result = C_NewAssociation (hSession, &hAssociation, &sigMechanism,
    sigKey, NULL_PTR, CK_TRUE);

// Verify received message in pieces

// first piece

result = C_ProcessMessage (hSession, hAssociation,
    NULL_PTR, 0, // no encrypt/decrypt so no input data
    associatedData1, associatedDataLen1,
    NULL_PTR, NULL_PTR, // no encrypt/decrypt so no output
    NULL_PTR, NULL_PTR, // Not last part so no signature
    NULL_PTR, 0, // no IV data,
    PHASE_FIRST);

// Last piece, emit the signature

result = C_ProcessMessage (hSession, hAssociation,
    NULL_PTR, 0, // no encrypt/decrypt so no input data
    associatedData2, associatedDataLen2,
    NULL_PTR, NULL_PTR, // no encrypt/decrypt so no output
    signature, &sigLen, // ptr because of dual use of param,
    NULL_PTR, 0, // no IV data,
    PHASE_FINAL);}
result = C_CloseAssociation (hSession, hAssociation);