# ANSI C Cryptographic API Profile
# for SHA-3 Candidate Algorithm Submissions

Revision 2: December 19, 2007

## 1. Overview

This document specifies the ANSI C interface profile for implementations of SHA-3 candidate algorithms. C implementations shall support the syntax and parameterization of the interface profile messages as described in this API. The API consists of one structure and 3 functions to manipulate the structure. The functions specified in this API have return values listed that are largely used to supply error codes in the event of incomplete execution of the routines. The error values listed are not meant to be an exhaustive list. If additional error codes are useful for your implementation, please provide them.

## 2. hashState

The *hashState* structure contains all information necessary to describe the current state of the SHA-3 candidate algorithm. The only required field, *hashbitlen*, indicates the output size of this particular instantiation of the hash algorithm. Algorithm specific fields follow the comment in the structure below. These include things like data storage needed to hold intermediate values, tables, unprocessed data, etc. **All implementations must be sure to document any algorithm-specific parameters and their use.**

```
typedef struct {
     int  hashbitlen;
     /* The following parameters are algorithm specific */
     } hashState;
```

## 3. Initization

Each SHA-3 submitter will be required to implement this interface because NIST anticipates that some candidate algorithms will have unique requirements to initialize the *hashState* structure.

The ANSI C programming interface uses a function called *Init()* to initialize the *hashState* structure. As stated above, the *hashState* structure contains the *hashbitlen* of this particular instantiation, as well as any algorithm specific parameters needed. Implementations shall support, at minimum, *hashbitlen* values of 224, 256, 384, and 512-bits. Additionally, if an algorithm can support other hash lengths, these should be supported in the code as well.

The initialization function, *Init(),* is called with the appropriate parameters which get loaded into the *hashState* structure. These parameters are then used to perform any data independent setup that is necessary, e.g., initialization of any intermediate values, initialization of any tables, etc.

❖ **Init()**

```
int Init(hashState *state, int hashbitlen)
```

Initializes a *hashState* with the output hash length of this particular instantiation. Additionally, any data independent setup is performed.

**Parameters**:
`state`: a structure that holds the *hashState* information
`hashbitlen`: an integer value that indicates the length of the hash output in bits.

**Returns**:
`SUCCESS - on success`
`BAD_HASHBITLEN - hashbitlen is invalid (e.g., unknown value)`

## 4. Update

The ANSI C programming interface uses a function called *Update()* to process data using the algorithm's compression function. Whatever integral amount of data the *Update()* routine can process through the compression function is handled. Any remaining data must be stored for future processing. For example, SHA-1 has an internal structure of 512-bit data blocks. If the *Update()* function is called with 768-bits of data the first 512-bits will be processed through the compression function (with appropriate updating of the chaining values) and 256-bits will be retained for future processing. If 2048-bits of data were provided, all 2048-bits would be processed immediately.

The *Update()* function is called with a pointer to the appropriate *hashState* structure, the *data* to be processed, the length of the data to be processed (*datalen*), and an *offset* pointer to locate the data within the entire message. The *Update()* routine processes as much data as it can, updating all appropriate intermediate values, and returns a status code. The *offset* parameter has been provided to facilitate various designs. In particular it may be useful for parallelizable algorithms.

❖ **Update()**

```
int Update(hashState *state, unsigned char *data, int datalen, int offset)
```

Process the supplied data.

**Parameters**:
`state`: a structure that holds the *hashState* information
`data`: the data to be hashed
`datalen`: the length, in bits, of the data to be hashed
`offset`: the offset, in bytes, of the data to be hashed within the entire message

**Returns**:
`SUCCESS - on success`

## 5. Finalization

The ANSI C programming interface uses a function called *Final()* to process any remaining partial block of data and to perform any output filtering that may be needed to produce the final hash value. For example, SHA-1 requires appending a "1"-bit to the end of the message followed by an appropriate number of "0"-bits and the length field. This is all processed through the compression function to produce the final hash value for the message.

The *Final()* function is called with pointers to the appropriate *hashState* structure and the storage for the final hash value to be returned (*hashval*). The *Final()* routine performs any post processing that is necessary, including the handling of any partial blocks, and places the final hash value in *hashval*. Lastly, an appropriate status value is returned.

### ❖ Final()

```
int Final(hashState *state, unsigned char *hashval)
```

Perform any post processing and output filtering required and return the final hash value.

**Parameters**:
`state`: a structure that holds the *hashState* information
`hashval`: the storage for the final hash value to be returned

**Returns**:
`SUCCESS - on success`

## 6. Additional Information

Return Values:

```
SUCCESS             0
BAD_HASHBITLEN      1
/*  Additional user defined return values */
```

Sample *hashState* for SHA-1:

```
typedef struct {
     int  hashbitlen;
     /* The following parameters are algorithm specific */

     // The following are the internal chaining values
     unsigned long       H[5];     // 5 * 32-bit word

     // The following is the internal 512-bit block
     unsigned long       W[16];    // 16 * 32-bit words

     // The following counts the amount of data processed for
     // post processing. It is the value placed in the last
     // 64-bits of the last block processed
     unsigned long       bits_processed[2];  //  64-bit counter

     // How much of W[] hasn't been processed yet
     // Needed by Final() to handle partial blocks
     int            unprocessed_bits;   // small counter <512
     } hashState;
```

A complete hash of a message can be obtained with the following function:

```
int
Hash(unsigned char *data, int datalen,
     unsigned char *hashval, int hashbitlen)
{
     hashState state;

     Init(&state, hashbitlen);
     Update(&state, data, datalen, 0);
     Final(&state, hashval);
}
```