Paper SAS2800-2016

Implementing Hashing Techniques in SAS®

Richard D. Langston, SAS Institute Inc., Cary, NC

ABSTRACT

Although hashing methods such as SHA256 and MD5 are already available in SAS®, users may need other methods. This presentation shows how hashing and methods can be implemented using SAS DATA steps and macros, with judicious use of the bitwise functions (BAND, BOR, and so on) and by referencing public domain sources.

INTRODUCTION

Hashing algorithms have been around for many years. In short, a hashing algorithm can accept an arbitrarily large stream of data and produce a relatively short fixed-length unique binary stream. Perhaps the best known, and an algorithm that has been around a long time, is CRC (cyclical redundancy check).

A resultant hash value might be used to confirm that the contents of a file is as expected after a download or some other data transfer. It might be used as input into a subsequent hashing or encryption algorithm.

Hashing algorithms have improved over time, in that they have fewer "collisions," where two different input streams result in the same hash. More recent algorithms include MD5, SHA-1, and SHA-256.

Some of these hashing algorithms are available within the SAS® language as functions. These include MD5 and SHA256.

Before the SHA256 function was added in SAS 9.4, I was asked to provide DATA step code for the algorithm. This paper describes what I did to provide this code. If you have the need for other algorithms, you may find my approach useful and you can model your implementation after mine.

THE BINARY FUNCTIONS

Before understanding about implementations for any hashing technique, you should first be sure to understand about some fundamental functions that will undoubtedly be used. These are the SAS binary functions: BAND, BOR, BXOR, BNOT, BLSHIFT, and BRSHIFT.

You may have seen these functions described in manuals in the past, and may have even used them over time. But most SAS users are likely unfamiliar with them.

These functions perform bitwise operations. For example, the BOR function performs a bitwise OR operation on two arguments that can be 1 or 0. Remember that OR-ing a 0 with 0 yields 0, but all other combinations yield 1. So BOR(0,0) results in 0, while BOR(0,1) results in 1. But be aware that because the function is bitwise, each pair of corresponding bits are OR-ed with each other. So BOR(3,4), where 3 is 011 in binary and 4 is 100 in binary, the result is 111, which is 7 in decimal. The BAND function performs bitwise AND operations, where only 1 and 1 yields 1, and any other combination yields 0. So BAND(3,4) would yield 0 because 011 AND-ed with 100 would be 000. BNOT performs bitwise NOT-ing where the bit is flipped. BXOR performs bitwise exclusive OR such that when both operands are the same the result is 0, and if they are different, the result is 1. For example, BXOR(1,0) yields 1 while BXOR(1,1) yields 0.

The BLSHIFT function shifts bits to the left and BRSHIFT shifts bits to the right. Any bits "shifted off the end" go into "the bit bucket" and are lost. For example, BRSHIFT(3,1) shifts the bits 11 to the right 1 bit, so the rightmost 1 is lost and the result is 1.

Note that all of these binary functions are limited to 32 bits.

THE SHA-256 ALGORITHM

The SHA-256 algorithm is public domain, having been originally developed by the National Security Agency. It is described in many different Internet sources, such as on Wikipedia.

There have also been a number of available implementations of SHA-256 and its companions.

When I was asked to implement SHA-256 in DATA step SAS code, I found a suitable implementation of the algorithm in the C language (which happens to be a primary implementation language for the SAS System). This source code is implemented by Mr. Oliver Gay and can be found at http://www.ouah.org/ogay/sha2/. My intention was to translate the C code to DATA step code with minimal change, and preserve the algorithm in the process.

Fortunately, the SAS DATA step language and the C language have some similarities, such as the use of semicolons to end statements, /* */ comments, square brackets for array references, etc. So much of the source code could be used as is.

An exception was the use of all the logical operations, which are heavily used in any hashing algorithm. Although the BAND, BOR, etc., functions would perform the logical operations, the operation symbols in the C language are different: AND is &, OR is |, NOT is ~, exclusive OR is ^, left shift is <<, right shift is >>. Also, the MOD operation is %. So all such operations would need C macro substitutions to the SAS equivalent in order to migrate the C code to SAS code. (Note that although & and | are valid operations in SAS, they have different meanings in the C language. & and | are equivalent to && and || in C.)

MIGRATING THE C CODE TO DATA STEP CODE

I wanted to ensure that the C code still functioned as I changed the code to look more like SAS code. This way my migration would be less painful. So I added the following definitions to the C code, so that all subsequent references to logical operators would be using the SAS equivalents. And all the special definitions in the C code were revised to use the definitions too.

```
/*-----SAS-like definitions added-----*/
#define BRSHIFT(x,y) ((x) >> (y))
#define BLSHIFT(x,y) ((x) << (y))
#define BAND(x,y) ((x) & (y))
#define BOR(x,y)  ((x) | (y))
#define BXOR(x,y) ((x) ^ (y))
#define BNOT(x)   (~(x))
#define MOD(x,y)  ((x) % (y))
#define IFC(c,t,f) ((c) ? (t) : (f))

#define SIZEOFXSL3 32

/*-----definitions changed to use above functions-----*/
#if 0
#define SHFR(x, n)    (x >> n)
#define ROTR(x, n)   ((x >> n) | (x << (SIZEOFXSL3 - n)))
#define CH(x, y, z)  ((x & y) ^ (~x & z))
#define MAJ(x, y, z) ((x & y) ^ (x & z) ^ (y & z))
#define SHA256_F1(x) (ROTR(x,  2) ^ ROTR(x, 13) ^ ROTR(x, 22))
#define SHA256_F2(x) (ROTR(x,  6) ^ ROTR(x, 11) ^ ROTR(x, 25))
#define SHA256_F3(x) (ROTR(x,  7) ^ ROTR(x, 18) ^ SHFR(x,  3))
#define SHA256_F4(x) (ROTR(x, 17) ^ ROTR(x, 19) ^ SHFR(x, 10))
#else
#define SHFR(x, n)   BRSHIFT(x,n)
#define ROTR(x, n)   BOR(BRSHIFT(x,n),BLSHIFT(x,SIZEOFXSL3 - n))
#define CH(x, y, z)  BXOR(BAND(x,y),BAND(BNOT(x),z))
#define MAJ(x, y, z) BXOR(BXOR(BAND(x,y),BAND(x,z)),BAND(y,z))
```

```
#define SHA256_F1(x) BXOR(BXOR(ROTR(x,  2),ROTR(x, 13)),ROTR(x, 22))
#define SHA256_F2(x) BXOR(BXOR(ROTR(x,  6),ROTR(x, 11)),ROTR(x, 25))
#define SHA256_F3(x) BXOR(BXOR(ROTR(x,  7),ROTR(x, 18)),SHFR(x,  3))
#define SHA256_F4(x) BXOR(BXOR(ROTR(x, 17),ROTR(x, 19)),SHFR(x, 10))
#endif
```

I could then test the C code with these revisions to ensure it still worked properly. Once I confirmed that, I could migrate the C code over to the corresponding SAS code.

THE SAS CODE

This is what I wanted the final SAS code to look like. An account number of length 19 is provided, along with the proper SHA-256 digest, which is a 32-byte field represented in 64 hex characters. This digest should match with the one returned from the SHA-256 algorithm. The algorithm would be encapsulated in a LINK routine, and the code would be expanded from a macro reference.

```
data _null_;
     length acctnum $19 acctnum_sha256 should_be $32;
     input acctnum: $char19. should_be: $hex64.;
     link sha256;
     if acctnum_sha256=should_be
        then status='correct  ';
     else status='incorrect';
     put    acctnum= / acctnum_sha256=$hex64. /
         +5 should_be=$hex64. / status=;
     return;
sha256:;
     %sha256_link_routine;
     return;
     cards;
1234567890123456789
22085aa929bcd7af4b23d9d9c046a1d4fde8be51f79d91392efafef96574ab01
1111111111111111111
b4db3abdc9d8c88b51aac2b2d3d14e74678b32ef413a9354911109e9cfa3d2e9
run;
```

The revision of the original source code appears below as an appendix, followed by the SAS macro definition for %SHA256_LINK_ROUTINE derived from that revision.

CONCLUSION

The SAS binary functions are very useful for hashing techniques, and can be used in your own implementation of public algorithms. If these algorithms are in other languages such as C, you can migrate the implementation to SAS using the technique I have described where you substitute the use of the binary functions.

REFERENCES

Gay, Oliver. 2007. "SHA-224, SHA-256, SHA-384 and SHA-512." http://www.ouah.org/ogay/sha2/. Accessed February 2016.

Hash'em all! http://www.hashemall.com. Accessed February 2016.

"SHA-2." Wikipedia. https://en.wikipedia.org/wiki/SHA-2. Accessed February 2016.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Richard D. Langston
SAS Institute Inc.
rick.langston@sas.com

APPENDIX I: C CODE IMPLEMENTATION - REVISED

```
/* This is the original source as written by Mr. Oliver Gay, but
   with the following changes:
   * header comment removed for brevity
   * added necessary code from sha2.h so the source is self-contained
   * removed all references to SHA512, SHA338, SHA224, SHA384,
   * removed all references to PACK64/UNPACK64
   * removed all UNROLL_LOOPS references
   * removed code to handle message size over 64 bytes to simplify
     the code for demonstration purposes, and removed the tests for
     longer messages.
   * revised to use the SAS function equivalents to simpler migration
     To SAS code
 */
#include <string.h>

#define SHA256_DIGEST_SIZE ( 256 / 8)
#define SHA256_BLOCK_SIZE  ( 512 / 8)

typedef unsigned char uint8;
typedef unsigned int  uint32;
typedef unsigned long long uint64;

typedef struct {
    unsigned int tot_len;
    unsigned int len;
    unsigned char block[2 * SHA256_BLOCK_SIZE];
    uint32 h[8];
} sha256_ctx;

typedef sha256_ctx sha224_ctx;

void sha256_init(sha256_ctx * ctx);
void sha256_update(sha256_ctx *ctx, const unsigned char *message,
                   unsigned int len);
void sha256_final(sha256_ctx *ctx, unsigned char *digest);
void sha256(const unsigned char *message, unsigned int len,
            unsigned char *digest);
```

```
/*-----SAS-like definitions added-----*/
#define BRSHIFT(x,y) ((x) >> (y))
#define BLSHIFT(x,y) ((x) << (y))
#define BAND(x,y) ((x) & (y))
#define BOR(x,y)  ((x) | (y))
#define BXOR(x,y) ((x) ^ (y))
#define BNOT(x)   (~(x))
#define MOD(x,y)  ((x) % (y))
#define IFC(c,t,f) ((c) ? (t) : (f))

#define SIZEOFXSL3 32

/*-----definitions changed to use above functions-----*/
#if 0
#define SHFR(x, n)    (x >> n)
#define ROTR(x, n)   ((x >> n) | (x << (SIZEOFXSL3 - n)))
#define CH(x, y, z)  ((x & y) ^ (~x & z))
#define MAJ(x, y, z) ((x & y) ^ (x & z) ^ (y & z))
#define SHA256_F1(x) (ROTR(x,  2) ^ ROTR(x, 13) ^ ROTR(x, 22))
#define SHA256_F2(x) (ROTR(x,  6) ^ ROTR(x, 11) ^ ROTR(x, 25))
#define SHA256_F3(x) (ROTR(x,  7) ^ ROTR(x, 18) ^ SHFR(x,  3))
#define SHA256_F4(x) (ROTR(x, 17) ^ ROTR(x, 19) ^ SHFR(x, 10))
#else
#define SHFR(x, n)   BRSHIFT(x,n)
#define ROTR(x, n)   BOR(BRSHIFT(x,n),BLSHIFT(x,SIZEOFXSL3 - n))
#define CH(x, y, z)  BXOR(BAND(x,y),BAND(BNOT(x),z))
#define MAJ(x, y, z) BXOR(BXOR(BAND(x,y),BAND(x,z)),BAND(y,z))
#define SHA256_F1(x) BXOR(BXOR(ROTR(x,  2),ROTR(x, 13)),ROTR(x, 22))
#define SHA256_F2(x) BXOR(BXOR(ROTR(x,  6),ROTR(x, 11)),ROTR(x, 25))
#define SHA256_F3(x) BXOR(BXOR(ROTR(x,  7),ROTR(x, 18)),SHFR(x,  3))
#define SHA256_F4(x) BXOR(BXOR(ROTR(x, 17),ROTR(x, 19)),SHFR(x, 10))
#endif

#define UNPACK32(x, str)                        \
{                                               \
    *((str) + 3) = (uint8) ((x)      );         \
    *((str) + 2) = (uint8) ((x) >>  8);         \
    *((str) + 1) = (uint8) ((x) >> 16);         \
    *((str) + 0) = (uint8) ((x) >> 24);         \
}

#define PACK32(str, x)                          \
{                                               \
    *(x) =   ((uint32) *((str) + 3)      )      \
           | ((uint32) *((str) + 2) <<  8)      \
           | ((uint32) *((str) + 1) << 16)      \
           | ((uint32) *((str) + 0) << 24);     \
}

/* Macros used for loops unrolling */

#define SHA256_SCR(i)                           \
```

```
{                                                   \
    w[i] =  SHA256_F4(w[i -  2]) + w[i -  7]  \
          + SHA256_F3(w[i - 15]) + w[i - 16]; \
}


#define SHA256_EXP(a, b, c, d, e, f, g, h, j)               \
{                                                           \
    t1 = wv[h] + SHA256_F2(wv[e]) + CH(wv[e], wv[f], wv[g]) \
         + sha256_k[j] + w[j];                              \
    t2 = SHA256_F1(wv[a]) + MAJ(wv[a], wv[b], wv[c]);       \
    wv[d] += t1;                                            \
    wv[h] = t1 + t2;                                        \
}



uint32 sha256_h0[8] =
            {0x6a09e667, 0xbb67ae85, 0x3c6ef372, 0xa54ff53a,
             0x510e527f, 0x9b05688c, 0x1f83d9ab, 0x5be0cd19};


uint32 sha256_k[64] =
            {0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5,
             0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5,
             0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3,
             0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174,
             0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc,
             0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da,
             0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7,
             0xc6e00bf3, 0xd5a79147, 0x06ca6351, 0x14292967,
             0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13,
             0x650a7354, 0x766a0abb, 0x81c2c92e, 0x92722c85,
             0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3,
             0xd192e819, 0xd6990624, 0xf40e3585, 0x106aa070,
             0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5,
             0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f, 0x682e6ff3,
             0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208,
             0x90befffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2};


/* SHA-256 functions */

void sha256_transf(sha256_ctx *ctx, const unsigned char *message,
                   unsigned int block_nb)
{
    uint32 w[64];
    uint32 wv[8];
    uint32 t1, t2;
    const unsigned char *sub_block;
    int i;
```

```
    int j;

    for (i = 0; i < (int) block_nb; i++) {
        sub_block = message + BLSHIFT(i,6);

        for (j = 0; j < 16; j++) {
            PACK32(&sub_block[BLSHIFT(j,2)], &w[j]);
        }

        for (j = 16; j < 64; j++) {
            SHA256_SCR(j);
        }

        for (j = 0; j < 8; j++) {
            wv[j] = ctx->h[j];
        }

        for (j = 0; j < 64; j++) {
            t1 = wv[7] + SHA256_F2(wv[4]) + CH(wv[4], wv[5], wv[6])
                + sha256_k[j] + w[j];
            t2 = SHA256_F1(wv[0]) + MAJ(wv[0], wv[1], wv[2]);
            wv[7] = wv[6];
            wv[6] = wv[5];
            wv[5] = wv[4];
            wv[4] = wv[3] + t1;
            wv[3] = wv[2];
            wv[2] = wv[1];
            wv[1] = wv[0];
            wv[0] = t1 + t2;
        }

        for (j = 0; j < 8; j++) {
            ctx->h[j] += wv[j];
        }
    }
}

void sha256(const unsigned char *message, unsigned int len, unsigned
char *digest)
{
    sha256_ctx ctx;

    sha256_init(&ctx);
    sha256_update(&ctx, message, len);
    sha256_final(&ctx, digest);
}

void sha256_init(sha256_ctx *ctx)
{
    int i;
    for (i = 0; i < 8; i++) {
        ctx->h[i] = sha256_h0[i];
```

```
    }

    ctx->len = 0;
    ctx->tot_len = 0;
}

void sha256_update(sha256_ctx *ctx, const unsigned char *message,
                   unsigned int len)
{
    unsigned int block_nb;
    unsigned int new_len, rem_len, tmp_len;
    const unsigned char *shifted_message;

    tmp_len = SHA256_BLOCK_SIZE - ctx->len;
    rem_len = IFC(len < tmp_len, len , tmp_len);

    memcpy(&ctx->block[ctx->len], message, rem_len);

    if (ctx->len + len < SHA256_BLOCK_SIZE) {
        ctx->len += len;
        return;
    }

    new_len = len - rem_len;
    block_nb = new_len / SHA256_BLOCK_SIZE;

    shifted_message = message + rem_len;

    sha256_transf(ctx, ctx->block, 1);
    sha256_transf(ctx, shifted_message, block_nb);

    rem_len = MOD(new_len,SHA256_BLOCK_SIZE);

    memcpy(ctx->block, &shifted_message[block_nb << 6],
           rem_len);

    ctx->len = rem_len;
    ctx->tot_len += BLSHIFT((block_nb + 1),6);
}

void sha256_final(sha256_ctx *ctx, unsigned char *digest)
{
    unsigned int block_nb;
    unsigned int pm_len;
    unsigned int len_b;

    int i;

    block_nb = (1 + ((SHA256_BLOCK_SIZE - 9)
                     < MOD(ctx->len,SHA256_BLOCK_SIZE)));
```

```
    len_b = BLSHIFT((ctx->tot_len + ctx->len),3);
    pm_len = BLSHIFT(block_nb,6);

    memset(ctx->block + ctx->len, 0, pm_len - ctx->len);
    ctx->block[ctx->len] = 0x80;
    UNPACK32(len_b, ctx->block + pm_len - 4);

    sha256_transf(ctx, ctx->block, block_nb);

    for (i = 0 ; i < 8; i++) {
        UNPACK32(ctx->h[i], &digest[BLSHIFT(i,2)]);
    }
}

#include <stdio.h>
#include <stdlib.h>

void test(const unsigned char *vector, unsigned char *digest,
          unsigned int digest_size)
{
    unsigned char output[2 * SHA256_DIGEST_SIZE + 1];
    int i;

    output[2 * digest_size] = '\0';

    for (i = 0; i < (int) digest_size ; i++) {
       sprintf((char *) output + 2 * i, "%02x", digest[i]);
    }

    printf("H: %s\n", output);
    if (strcmp((char *) vector, (char *) output)) {
        fprintf(stderr, "Test failed.\n");
        exit(EXIT_FAILURE);
    }
}

int main()
{
    char digest[SHA256_DIGEST_SIZE];
    static const unsigned char message1[] = "12345678901234567890";
    static const unsigned char message2[] = "11111111111111111111";

    sha256(message1, strlen((char *) message1), digest);

test("22085aa929bcd7af4b23d9d9c046a1d4fde8be51f79d91392efafef96574ab01
",digest, SHA256_DIGEST_SIZE);
    sha256(message2, strlen((char *) message2), digest);

test("b4db3abdc9d8c88b51aac2b2d3d14e74678b32ef413a9354911109e9cfa3d2e9
",digest, SHA256_DIGEST_SIZE);

    printf("\n");
```

```
    printf("All tests passed.\n");

    return 0;
}
```

APPENDIX II: SAS CODE IMPLEMENTATION

```
/* Code converted to SAS, as the sha256_link_routine macro. */

%macro sha256_link_routine;

%let SHA256_BLOCK_SIZE = 64;
%let SHA256_BLOCK_SIZE_TIMES2 = 128;

%let acctnum_length = 19;

/* SAS uses 1-based indexing while C uses 0-based indexing, so these
   macro variables allow for the code to remain more similar. */
%let _0 = 1;
%let _1 = 2;
%let _2 = 3;
%let _3 = 4;
%let _4 = 5;
%let _5 = 6;
%let _6 = 7;
%let _7 = 8;

%macro  SHFR(x, n);    BRSHIFT(&x,&n) %mend;
%macro  ROTR(x, n);    BOR(BRSHIFT(&x,&n),BLSHIFT(&x,32 - &n)) %mend;
%macro  CH(x, y, z);   BXOR(BAND(&x,&y),BAND(BNOT(&x),&z)) %mend;
%macro  MAJ(x, y, z); BXOR(BXOR(BAND(&x,&y),BAND(&x,&z)),BAND(&y,&z))
%mend;
%macro  SHA256_F1(x); BXOR(BXOR(%ROTR(&x,  2),%ROTR(&x, 13)),%ROTR(&x,
22)) %mend;
%macro  SHA256_F2(x); BXOR(BXOR(%ROTR(&x,  6),%ROTR(&x, 11)),%ROTR(&x,
25)) %mend;
%macro  SHA256_F3(x); BXOR(BXOR(%ROTR(&x,  7),%ROTR(&x, 18)),%SHFR(&x,
3)) %mend;
%macro  SHA256_F4(x); BXOR(BXOR(%ROTR(&x, 17),%ROTR(&x, 19)),%SHFR(&x,
10)) %mend;

%macro SHA256_SCR(i);
    w[&i] =  mod(%SHA256_F4(w[&i -  2]) + w[&i -  7]
              +  %SHA256_F3(w[&i - 15]) + w[&i - 16],100000000x);
%mend;

array sha256_h0[8] _temporary_ (
            06a09e667x, 0bb67ae85x, 03c6ef372x, 0a54ff53ax,
```

```
              0510e527fx, 09b05688cx, 01f83d9abx, 05be0cd19x
              );

array sha256_k[64] _temporary_ (
              0428a2f98x, 071374491x, 0b5c0fbcfx, 0e9b5dba5x,
              03956c25bx, 059f111f1x, 0923f82a4x, 0ab1c5ed5x,
              0d807aa98x, 012835b01x, 0243185bex, 0550c7dc3x,
              072be5d74x, 080deb1fex, 09bdc06a7x, 0c19bf174x,
              0e49b69c1x, 0efbe4786x, 00fc19dc6x, 0240ca1ccx,
              02de92c6fx, 04a7484aax, 05cb0a9dcx, 076f988dax,
              0983e5152x, 0a831c66dx, 0b00327c8x, 0bf597fc7x,
              0c6e00bf3x, 0d5a79147x, 006ca6351x, 014292967x,
              027b70a85x, 02e1b2138x, 04d2c6dfcx, 053380d13x,
              0650a7354x, 0766a0abbx, 081c2c92ex, 092722c85x,
              0a2bfe8a1x, 0a81a664bx, 0c24b8b70x, 0c76c51a3x,
              0d192e819x, 0d6990624x, 0f40e3585x, 0106aa070x,
              019a4c116x, 01e376c08x, 02748774cx, 034b0bcb5x,
              0391c0cb3x, 04ed8aa4ax, 05b9cca4fx, 0682e6ff3x,
              0748f82eex, 078a5636fx, 084c87814x, 08cc70208x,
              090befffax, 0a4506cebx, 0bef9a3f7x, 0c67178f2x
              );

array w[64] _temporary_;
array wv[8] _temporary_;
array ctx_h[8] _temporary_;
length ctx_len ctx_tot_len 8;
length ctx_block $&SHA256_BLOCK_SIZE_TIMES2.;

length message $64;

     link sha256_init;
     message = acctnum;
     len = &acctnum_length;
     link sha256_update;
     link sha256_final;
     return;

sha256_transf:;
     return;

sha256_init:;
     do j=1 to 8;
        ctx_h[j] = sha256_h0[j];
        end;
     ctx_len = 0;
     ctx_tot_len = 0;
     return;


sha256_update:;
     tmp_len = &SHA256_BLOCK_SIZE - ctx_len;
     rem_len = ifn(len < tmp_len,len,tmp_len);
```

```
     substr(ctx_block,ctx_len+1,rem_len) = message;

     if (ctx_len + len < &SHA256_BLOCK_SIZE) then do;
        ctx_len + len;
        return;
        end;

     start = 1; block_nb = 1;
     link sha256_transf;

     new_len = len - rem_len;
     block_nb = new_len / &SHA256_BLOCK_SIZE;
     start = rem_len+1;
     link sha256_transf;

     rem_len = MOD(new_len,&SHA256_BLOCK_SIZE) ;

     return;


sha256_final:;

     block_nb = (1 + ((&SHA256_BLOCK_SIZE - 9)
                    < mod(ctx_len , &SHA256_BLOCK_SIZE)));

     len_b = BLSHIFT((ctx_tot_len + ctx_len),3);
     pm_len = blshift(block_nb,6);


        message = acctnum || '80'x ||
                  repeat('00'x,56-(&acctnum_length+1)-1) ||
                  put(&acctnum_length.*8,s370fpib8.);

        i = 1;
        do j=1 to 16;
           w[j] = input(substr(message,i,4),s370fpib4.); i+4;
           end;

        do j=17 to 64; %SHA256_SCR(j); end;

        do j=1 to 8; wv[j] = ctx_h[j]; end;

        do j=1 to 64;
           t1 = wv[&_7] +
                %SHA256_F2(wv[&_4]) + %CH(wv[&_4], wv[&_5], wv[&_6]) +
                sha256_k[j] + w[j];
           t2 = %SHA256_F1(wv[&_0]) + %MAJ(wv[&_0], wv[&_1], wv[&_2]);
            wv[&_7] = wv[&_6];
            wv[&_6] = wv[&_5];
            wv[&_5] = wv[&_4];
            wv[&_4] = mod(wv[&_3] + t1,100000000x);
```

```
            wv[&_3] = wv[&_2];
            wv[&_2] = wv[&_1];
            wv[&_1] = wv[&_0];
            wv[&_0] = mod(t1 + t2,100000000x);
            end;

        do j=1 to 8; ctx_h[j] + wv[j]; end;

        i=1;
        do j=1 to 8;
           substr(acctnum_sha256,i,4) =
             put(mod(sha256_h0[j] + wv[j],100000000x),s370fpib4.);
           i+4;
           end;
     return;
%mend;
```