Licence L2 STS Mention SPI Parcours Informatique
Unité 174EN007
Sécurité Informatique

TP2
Calcul d'empreinte avec SHA512

```
#ifndef SHA512_H
#define SHA512_H

/**************************** DATA TYPES ****************************/
typedef unsigned char BYTE; // 8-bit byte
typedef unsigned long WORD; // 64-bit word

typedef struct {
	BYTE data[128];
	WORD datalen;
	unsigned long long bitlen_h;
	unsigned long long bitlen_l;
	WORD state[8];
} SHA512_CTX;

/*********************** FUNCTION DECLARATIONS **********************/
void sha512_compress(SHA512_CTX *ctx);
void sha512_init(SHA512_CTX *ctx);
void sha512_compute(SHA512_CTX *ctx, const BYTE data[], size_t len);
void sha512_convert(SHA512_CTX *ctx, BYTE hash[]);

#endif   // SHA512_H
```

```
/*************************** HEADER FILES ***************************/
#include <stdlib.h>
#include <stddef.h>
#include <memory.h>
#include "sha512.h"

/****************************** MACROS ******************************/
#define SHR(a,b) ((a) >> (b))
#define ROTR(a,b) (((a) >> (b)) | ((a) << (64 - (b))))

#define CH(x,y,z) (((x) & (y)) ^ (~(x) & (z)))
#define MAJ(x,y,z) (((x) & (y)) ^ ((x) & (z)) ^ ((y) & (z)))
#define EP0(x) (ROTR(x,28) ^ ROTR(x,34) ^ ROTR(x,39))
#define EP1(x) (ROTR(x,14) ^ ROTR(x,18) ^ ROTR(x,41))
#define SIG0(x) (ROTR(x,1) ^ ROTR(x,8) ^ SHR(x, 7))
#define SIG1(x) (ROTR(x,19) ^ ROTR(x,61) ^ SHR(x, 6))

/**************************** VARIABLES *****************************/
static const WORD k[80] = {
	0x428a2f98d728ae22, 0x7137449123ef65cd, 0xb5c0fbcfec4d3b2f, 0xe9b5dba58189dbbc,
	0x3956c25bf348b538, 0x59f111f1b605d019, 0x923f82a4af194f9b, 0xab1c5ed5da6d8118,
	0xd807aa98a3030242, 0x12835b0145706fbe, 0x243185be4ee4b28c, 0x550c7dc3d5ffb4e2,
	0x72be5d74f27b896f, 0x80deb1fe3b1696b1, 0x9bdc06a725c71235, 0xc19bf174cf692694,
	0xe49b69c19ef14ad2, 0xefbe4786384f25e3, 0x0fc19dc68b8cd5b5, 0x240ca1cc77ac9c65,
	0x2de92c6f592b0275, 0X4a7484aa6ea6e483, 0x5cb0a9dcbd41fbd4, 0x76f988da831153b5,
	0x983e5152ee66dfab, 0xa831c66d2db43210, 0xb00327c898fb213f, 0xbf597fc7beef0ee4,
	0xc6e00bf33da88fc2, 0xd5a79147930aa725, 0x06ca6351e003826f, 0x142929670a0e6e70,
	0x27b70a8546d22ffc, 0x2e1b21385c26c926, 0x4d2c6dfc5ac42aed, 0x53380d139d95b3df,
	0x650a73548baf63de, 0x766a0abb3c77b2a8, 0x81c2c92e47edaee6, 0x92722c851482353b,
	0xa2bfe8a14cf10364, 0xa81a664bbc423001, 0xc24b8b70d0f89791, 0xc76c51a30654be30,
	0xd192e819d6ef5218, 0xd69906245565a910, 0xf40e35855771202a, 0x106aa07032bbd1b8,
	0x19a4c116b8d2d0c8, 0x1e376c085141ab53, 0x2748774cdf8eeb99, 0x34b0bcb5e19b48a8,
	0x391c0cb3c5c95a63, 0x4ed8aa4ae3418acb, 0x5b9cca4f7763e373, 0x682e6ff3d6b2b8a3,
	0x748f82ee5defb2fc, 0x78a5636f43172f60, 0x84c87814a1f0ab72, 0x8cc702081a6439ec,
	0x90befffa23631e28, 0xa4506cebde82bde9, 0xbef9a3f7b2c67915, 0xc67178f2e372532b,
	0xca273eceea26619c, 0xd186b8c721c0c207, 0xeada7dd6cde0eb1e, 0xf57d4f7fee6ed178,
	0x06f067aa72176fba, 0x0a637dc5a2c898a6, 0x113f9804bef90dae, 0x1b710b35131c471b,
	0x28db77f523047d84, 0x32caab7b40c72493, 0x3c9ebe0a15c9bebc, 0x431d67c49c100d4c,
	0x4cc5d4becb3e42b6, 0x597f299cfc657e2a, 0x5fcb6fab3ad6faec, 0x6c44198c4a475817
};
```

```
/********************** FUNCTION DEFINITIONS **********************/
void sha512_compress(SHA512_CTX *ctx)
{
	WORD a, b, c, d, e, f, g, h, i, j, t1, t2, m[80];

	for (i = 0, j = 0; i < 16; ++i, j += 8)
		m[i] = ((WORD)ctx->data[j] << 56)     | ((WORD)ctx->data[j + 1] << 48) | ((WORD)ctx->data[j + 2] << 40) |
		     ((WORD)ctx->data[j + 3] << 32) | ((WORD)ctx->data[j + 4] << 24) | ((WORD)ctx->data[j + 5] << 16) |
		     ((WORD)ctx->data[j + 6] << 8)  | ((WORD)ctx->data[j + 7]);
	for ( ; i < 80; ++i)
		m[i] = SIG1(m[i - 2]) + m[i - 7] + SIG0(m[i - 15]) + m[i - 16];

	a = ctx->state[0];
	b = ctx->state[1];
	c = ctx->state[2];
	d = ctx->state[3];
	e = ctx->state[4];
	f = ctx->state[5];
	g = ctx->state[6];
	h = ctx->state[7];

	for (i = 0; i < 80; ++i) {
		t1 = h + EP1(e) + CH(e,f,g) + k[i] + m[i];
		t2 = EP0(a) + MAJ(a,b,c);
		h = g;
		g = f;
		f = e;
		e = d + t1;
		d = c;
		c = b;
		b = a;
		a = t1 + t2;
	}

	ctx->state[0] += a;
	ctx->state[1] += b;
	ctx->state[2] += c;
	ctx->state[3] += d;
	ctx->state[4] += e;
	ctx->state[5] += f;
	ctx->state[6] += g;
	ctx->state[7] += h;
}

void sha512_init(SHA512_CTX *ctx)
{
	ctx->datalen = 0;
	ctx->bitlen_h = ctx->bitlen_l = 0;
	ctx->state[0] = 0x6a09e667f3bcc908;
	ctx->state[1] = 0xbb67ae8584caa73b;
	ctx->state[2] = 0x3c6ef372fe94f82b;
	ctx->state[3] = 0xa54ff53a5f1d36f1;
	ctx->state[4] = 0x510e527fade682d1;
	ctx->state[5] = 0x9b05688c2b3e6c1f;
	ctx->state[6] = 0x1f83d9abfb41bd6b;
	ctx->state[7] = 0x5be0cd19137e2179;
}

void sha512_compute(SHA512_CTX *ctx, const BYTE data[], size_t len)
{
	WORD i;

	for (i = 0; i < len; ++i) {
		ctx->data[ctx->datalen] = data[i];
		ctx->datalen++;
		if (ctx->datalen == 128) {
			sha512_compress(ctx);
			ctx->bitlen_l += 1024;
			if (ctx->bitlen_l == 0xFFFFFFFFFFFFFC00) {  // (0xFFFFFFFFFFFFFFFF / 1024) * 1024
				ctx->bitlen_h++;
				ctx->bitlen_l = 0x3FF; // 0xFFFFFFFFFFFFFFFF - 0xFFFFFFFFFFFFFC00
			}
			ctx->datalen = 0;
		}
```

```
	}

	i = ctx->datalen;

	// Pad whatever data is left in the buffer.
	if (ctx->datalen < 112) {
		ctx->data[i++] = 0x80;
		while (i < 112)
			ctx->data[i++] = 0x00;
	}
	else {
		ctx->data[i++] = 0x80;
		while (i < 128)
			ctx->data[i++] = 0x00;
		sha512_compress(ctx);
		memset(ctx->data, 0, 112);
	}

	// Append to the padding the total message's length in bits.

	ctx->bitlen_l += ctx->datalen * 8;

	ctx->data[127] = ctx->bitlen_l;
	ctx->data[126] = ctx->bitlen_l >> 8;
	ctx->data[125] = ctx->bitlen_l >> 16;
	ctx->data[124] = ctx->bitlen_l >> 24;
	ctx->data[123] = ctx->bitlen_l >> 32;
	ctx->data[122] = ctx->bitlen_l >> 40;
	ctx->data[121] = ctx->bitlen_l >> 48;
	ctx->data[120] = ctx->bitlen_l >> 56;

	ctx->data[119] = ctx->bitlen_h;
	ctx->data[118] = ctx->bitlen_h >> 8;
	ctx->data[117] = ctx->bitlen_h >> 16;
	ctx->data[116] = ctx->bitlen_h >> 24;
	ctx->data[115] = ctx->bitlen_h >> 32;
	ctx->data[114] = ctx->bitlen_h >> 40;
	ctx->data[113] = ctx->bitlen_h >> 48;
	ctx->data[112] = ctx->bitlen_h >> 56;

	sha512_compress(ctx);
}

void sha512_convert(SHA512_CTX *ctx, BYTE hash[])
{
	int i;

	// Since this implementation uses little endian byte ordering and SHA uses big endian,
	// reverse all the bytes when copying the final state to the output hash.
	for (i = 0; i < 8; ++i) {
		hash[i]      = (ctx->state[0] >> (56 - i * 8));
		hash[i + 8]  = (ctx->state[1] >> (56 - i * 8));
		hash[i + 16] = (ctx->state[2] >> (56 - i * 8));
		hash[i + 24] = (ctx->state[3] >> (56 - i * 8));
		hash[i + 32] = (ctx->state[4] >> (56 - i * 8));
		hash[i + 40] = (ctx->state[5] >> (56 - i * 8));
		hash[i + 48] = (ctx->state[6] >> (56 - i * 8));
		hash[i + 56] = (ctx->state[7] >> (56 - i * 8));
	}
}
```

```
#include <stdio.h>
#include <memory.h>
#include <string.h>
#include "sha512.h"

void print_hash(unsigned char hash[])
{
   int idx;
   for (idx=0; idx < 64; idx++)
      printf("%02x",hash[idx]);
   printf("\n");
}

int main()
```

```
{
	unsigned char text1[]={"01234567879"},
		text2[]={"012345678790123456787901234567879012345678790123456787901234567879"},
		hash[64];

	SHA512_CTX ctx;

	// Hash one
	sha512_init(&ctx);
	sha512_compute(&ctx,text1,strlen(text1));
	sha512_convert(&ctx,hash);
	print_hash(hash);

	// Hash two
	sha512_init(&ctx);
	sha512_compute(&ctx,text2,strlen(text2));
	sha512_convert(&ctx,hash);
	print_hash(hash);

	return 0;
}
```