The Implementation of Tiny Encryption Algorithm (TEA) on PIC18F4550 microcontroller

Edi Permadi
Electrical Engineering 2005
President University
edipermadi@gmail.com | http://edipermadi.wordpress.com

Abstract. We presented a way to implement Tiny Encryption Algorithm (TEA) using an 8-bit microcontroller PIC18F4550

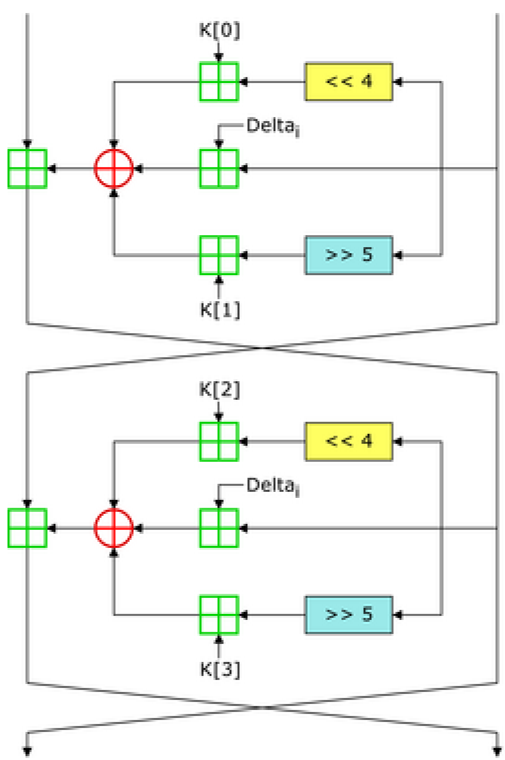
Introduction

Tiny Encryption Algorithm is a notable simple, fast and feistel based block cipher developed by David J. Wheeler and Roger M. Needham from Cambridge University. Tiny Encryption Algorithm has 32 rounds of simple processes which are shifts, additions and XORs. Tiny Encryption Algorithm has 128-bit key length and 64-bit block size.

TEA cipher key scheduling is simple anyway. It uses modulo 32-bit addition by delta (∂) constant. However, that constant is derived from the golden number as follow:

$$\partial = \left(\sqrt{5} - 1\right) \cdot 2^{31}$$

TEA cipher processes data block by block. Each block is consisted of two 32-bit half block. A half block is processed and swapped iteratively and all operations are performed on modulo 32-bit big endian manner. The detail of TEA cipher can be described as follow:



PIC18F4550 is an 8-bit microcontroller manufactured by Microchip Technology Inc. This microcontroller employs RISC architecture with native “carry enabled” adding instruction and “borrow enabled” subtracting instruction. Prior to algorithm implementation, PIC18F4550 has ability to cope such all requirements required by TEA cipher. Technically speaking, those 32-bit operations are available under emulation.

Implementation

Firstly, we discussed big-endian byte organization. The big-endian stated that 32-bit data is packed as 4 8-bit data where the least significant byte is located rightmost, in highest memory location and vice versa. For example 305419896_{10} will be represented as 0x12345678 where 0x12 located on the lower address of memory and 0x78 located on the higher address memory. In graphical representation, 0x12345678 is depected as follow.

addr + 0	addr + 1	addr + 2	addr + 3
0x12	0x34	0x56	0x78

Secondly, we emulated 32-bit operations in such simpler 8-bit operations that are primitive and native to PIC18F4550 microcontroller. The instruction emulation is done by combining 8-bit instruction to masquerade an expected 32-bit operation. Those emulated operations are shifting, XORing, adding and subtracting. Each of those operations is explained gradually below:

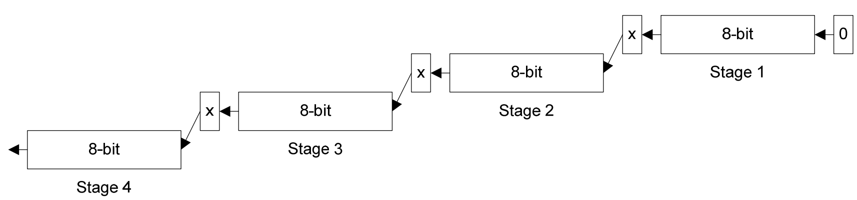
1. 32-bit Shifting

   32-bit shifting is done by chaining four “carry enabled” 8-bit instructions. By implementing this method, flowing bit is propagated from previous instruction to the current instruction through carry flag. However, it is necessary to reset carry bit before the first 8-bit shift to avoid unexpected result. The emulation detail is shown below.

   32-bit Shift Left

   The real 32-bit operation (left most bit overflow is ignored)

   Emulated 32-bit operation (left most bit overflow is ignored)

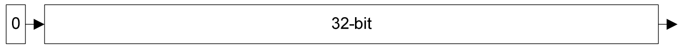


   The 32-bit shift left operation is emulated by a macro below

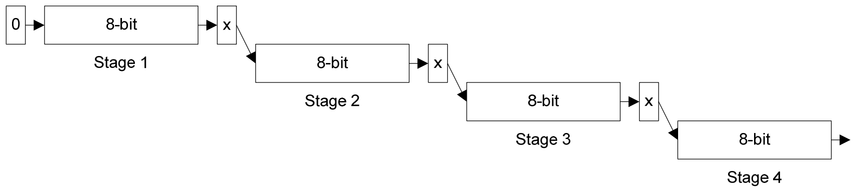
```
; 32-bit Shift Left Instruction emulation macro
shl32 MACRO arg
      bcf   status,c
      rlcf  arg+3,f
      rlcf  arg+2,f
      rlcf  arg+1,f
      rlcf  arg+0,f
      ENDM
```

32-bit Shift Right

The real 32-bit operation (right most bit overflow is ignored)



Emulated 32-bit operation (right most bit overflow is ignored)



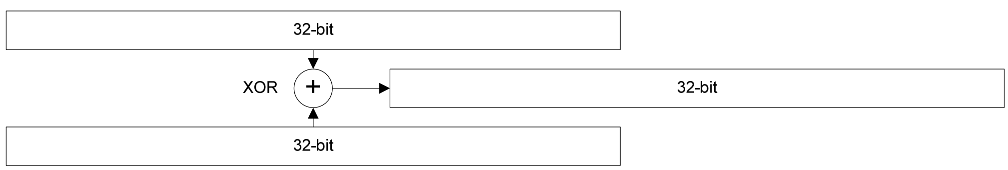
The 32-bit shift left operation is emulated by a macro below

```
; 32-bit Shift Right Instruction emulation macro
shr32 MACRO arg
      bcf   status,c
      rrcf  arg+0,f
      rrcf  arg+1,f
      rrcf  arg+2,f
      rrcf  arg+3,f
      ENDM
```

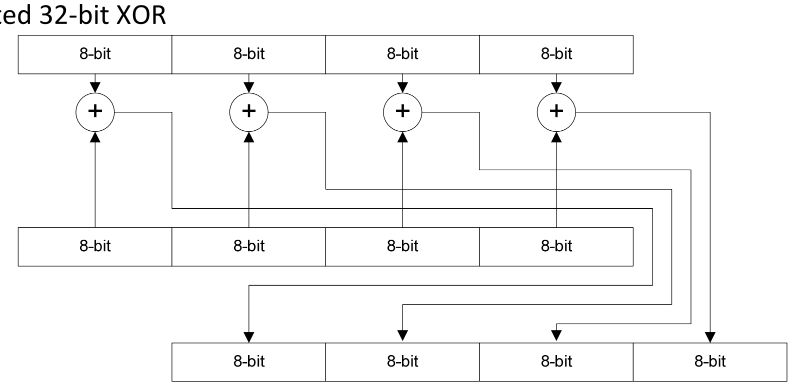
2. <u>32-bit bitwise XOR</u>

   32-bit bitwise XOR process is done by XORing two corresponding byte chunks. Sequential instruction order is optional, since each stage is independent to other and none is propagated from previous stage to the current stage.

The real 32-bit XOR



Emulated 32-bit XOR



The process above is represented as a MACRO below:

```
; 32-bit bitwise XOR emulation dst = (dst ^ src)
xor32 MACRO       dst, arg
      movf        src+0,w
      xorwf       dst+0,f
      movf        src+1,w
      xorwf       dst+1,f
      movf        src+2,w
      xorwf       dst+2,f
      movf        src+3,w
      xorwf       dst+3,f
      ENDM
```

3. <u>32-bit Addition / Subtraction</u>

   32-bit addition and subtraction has the same principal as 32-bit shifting implementation. 32-bit addition / subtraction is done by chaining 4 8-bit addition / subtraction in order of least significant byte to the most significant byte. Sequential order is important here since each stage is not independent to other. Carries and borrows are propagated from previous instruction to the current instruction, therefore “carry enabled” and “borrow enabled” instructions are employed here.

   The adding and subtracting emulation is done by macros below.

```
; 32-bit adding emulation dst = (dst + src)
add32 MACRO       dst, src
      movf        src+3,w
      addwf       dst+3,f
      movf        src+2,w
      addwfc      dst+2,f
      movf        src+1,w
      addwfc      dst+1,f
      movf        src+0,w
      addwfc      dst+0,f
      ENDM

; 32-bit subtracting emulation dst = (dst - src)
add32 MACRO       dst, src
      movf        src+3,w
      addwf       dst+3,f
      movf        src+2,w
      addwfc      dst+2,f
      movf        src+1,w
      addwfc      dst+1,f
      movf        src+0,w
      addwfc      dst+0,f
      ENDM
```

Due to performance issues, those macros above may subject to modification but the concept are still the same.

Thirdly, we discussed the C model of Tiny Encryption Algorithm (TEA) as a reference for assembly language reference. Both encryption and decryption routine of Tiny Encryption Algorithm has simple structure and independent from other inner functions. The encryption and decryption routine then shown below.

```c
#include <stdint.h>

// encryption routine
void encrypt (uint32_t* v, uint32_t* k) {
	uint32_t v0=v[0], v1=v[1], sum=0, i;
	uint32_t delta=0x9e3779b9;
	uint32_t k0=k[0], k1=k[1], k2=k[2], k3=k[3];
	for (i=0; i < 32; i++) {
		sum   += delta;
		v0    += ((v1<<4) + k0) ^ (v1 + sum) ^ ((v1>>5) + k1);
		v1    += ((v0<<4) + k2) ^ (v0 + sum) ^ ((v0>>5) + k3);
	}
	v[0]=v0; v[1]=v1;
}

// decryption routine
void decrypt (uint32_t* v, uint32_t* k) {
	uint32_t v0=v[0], v1=v[1], sum=0xC6EF3720, i;
	uint32_t delta=0x9e3779b9;
	uint32_t k0=k[0], k1=k[1], k2=k[2], k3=k[3];
	for (i=0; i<32; i++) {
		v1    -= ((v0<<4) + k2) ^ (v0 + sum) ^ ((v0>>5) + k3);
		v0    -= ((v1<<4) + k0) ^ (v1 + sum) ^ ((v1>>5) + k1);
		sum   -= delta;
	}
	v[0]=v0; v[1]=v1;
}
```

The assembly implementation of the code above is available below:

```
;===============================================================================
; Implementation of Tiny Encryption Algorithm (TEA) on PIC18F4550              ;
; Copyright (C) 2009 Edi Permadi                                               ;
;===============================================================================
;                                                                              ;
; Author              : Edi Permadi                                            ;
; Date Coded          : Jan 16, 2008                                           ;
; Version             : 1.0                                                    ;
; Last Modified       : Jan 16, 2008                                           ;
; downloaded from     : http://edipermadi.wordpress.com                        ;
; Email               : edipermadi@gmail.com                                   ;
;                                                                              ;
;===============================================================================
;                                                                              ;
; This program is free software: you can redistribute it and/or modify        ;
; it under the terms of the GNU General Public License as published by        ;
; the Free Software Foundation, either version 3 of the License, or            ;
; (at your option) any later version.                                          ;
;                                                                              ;
; This program is distributed in the hope that it will be useful,              ;
; but WITHOUT ANY WARRANTY; without even the implied warranty of               ;
; MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the                ;
; GNU General Public License for more details.                                 ;
;                                                                              ;
; You should have received a copy of the GNU General Public License            ;
; along with this program.  If not, see <http://www.gnu.org/licenses/>.        ;
;                                                                              ;
;===============================================================================
;
```

```
; Description
;       Tiny Encryption Algorithm (TEA) is a block cipher designed by Roger
; Needham and David Wheeler. This block cipher has 128-bit key length and 64-
; bit block length. This cipher employes 32 loops feistel structure. Each block
; of data is consisted of two 32 bit half block building.
;       TEA uses native and primitive 32 bit operations suchs shifts, adds, and
; bitwise XORs. Those instructions scalable and easy to implement in such lower
; computring environment, for instance microcontroller.
;       Thus, TEA is nothing than an easy and fast block cipher.
;
; Implementation
;       TEA is consisted of three 32-bit primitive instructions which are:
; addition, shift and XOR. Those 32-bit instructions were not directly
; applicable to PIC18F4550, thus the implementation is done by emulating native
; PIC18F4550 instructions to masquerade those 32-bit instructions. Those 32-bit
; instruction then elaborated step by step as follows:
;       1. 32-Bit Addition / Subtraction
;              The 32-bit additiona and subtraction is done by implementing
;              4 chained 8-bit additions in a macro. carry is propagated
;              through carry flag and implemented natively as add with carry
;              instruction. The implementation of 32-bit addition and
;              subtraction can be found inside this code as "add32" and
;              "sub32" macros.
;       2. 32-bit Bitwise XOR
;              The 32-bit bitwise XOR is easily done by implementing parallel 4
;              8-bit bitwise XOR. This instruction is implemented concurently
;              with 32 bit addition as "xadd32" and "xsub32" to avoid unused
;              cycles.
;       3. 32-Bit Shift Left / Right
;              The 32-bit shift is done by implementing sequential shift.
;              Bits are propagated through carry flag. Those implementations
;              are entitled "shr32" and "shl32".
;
; History
;       v1.0   Jan 16, 2009  Initial Release
;
; Benchmark
;       Encryprion : 6826 cycle
;       Decryption : 6830 cycle
;
; Test Vector
;       Plain  : 0x0123456789abcdef
;       Key    : 0x00112233445566778899aabbccddeeff
;       Cipher : 0x126c6b92c0653a3e
;



       LIST   P=PIC18F4550
       RADIX  DEC

;===============================================================================
; GPR Definition
;===============================================================================
k0     equ    0x00   ; key buffer
k1     equ    0x04
k2     equ    0x08
k3     equ    0x0c
v0     equ    0x10   ; half left side
v1     equ    0x14   ; half right side
sum    equ    0x18   ; summing buffer
cnt    equ    0x1c   ; loop counter
t0     equ    0x20   ; temporary register
```

```
t1     equ    0x24
t2     equ    0x28
t3     equ    0x2c




;===============================================================================
; SFR Definition
;===============================================================================
status equ    0x0fd8




;===============================================================================
; Bit Definition
;===============================================================================
z      equ    0x02
c      equ    0x00




;===============================================================================
; Useful Macros
;===============================================================================
; Load 8 bit value into a register
movlf  MACRO  reg,lit
       movlw  lit
       movwf  reg
       ENDM

; load integer type constant
movlf32       MACRO  reg,lit
       movlw  (lit >> .0 ) & 0xff
       movwf  reg+3
       movlw  (lit >> .8 ) & 0xff
       movwf  reg+2
       movlw  (lit >> .16) & 0xff
       movwf  reg+1
       movlw  (lit >> .24) & 0xff
       movwf  reg+0
       ENDM

; Add two integer
add32  MACRO  dst,src
       movf   src+3,w
       addwf  dst+3,f
       movf   src+2,w
       addwfc dst+2,f
       movf   src+1,w
       addwfc dst+1,f
       movf   src+0,w
       addwfc dst+0,f
       ENDM

; Add two integer
sub32  MACRO  dst,src
       movf   src+3,w
       subwf  dst+3,f
       movf   src+2,w
       subwfb dst+2,f
```

```
        movf   src+1,w
        subwfb dst+1,f
        movf   src+0,w
        subwfb dst+0,f
        ENDM

; Reset integer to zero
clrf32 MACRO  arg
        clrf   arg+0
        clrf   arg+1
        clrf   arg+2
        clrf   arg+3
        ENDM

; Rotate Right Integer
rrf32  MACRO  arg
        bcf    status,c
        rrcf   arg+0,f
        rrcf   arg+1,f
        rrcf   arg+2,f
        rrcf   arg+3,f
        ENDM

; Rotate Left Integer
rlf32  MACRO  arg
        bcf    status,c
        rlcf   arg+3,f
        rlcf   arg+2,f
        rlcf   arg+1,f
        rlcf   arg+0,f
        ENDM

; Add constant to integer
addl32 MACRO  arg,lit
        movlw  (lit >> .0 ) & 0xff
        addwf  arg+3,f
        movlw  (lit >> .8 ) & 0xff
        addwfc arg+2,f
        movlw  (lit >> .16) & 0xff
        addwfc arg+1,f
        movlw  (lit >> .24) & 0xff
        addwfc arg+0,f
        ENDM

; Subtract constant from integer
subl32 MACRO  arg,lit
        movlw  (lit >> .0 ) & 0xff
        subwf  arg+3,f
        movlw  (lit >> .8 ) & 0xff
        subwfb arg+2,f
        movlw  (lit >> .16) & 0xff
        subwfb arg+1,f
        movlw  (lit >> .24) & 0xff
        subwfb arg+0,f
        ENDM

; copy integer
mov32  MACRO  dst,src
        movff  src+0,dst+0
        movff  src+1,dst+1
        movff  src+2,dst+2
        movff  src+3,dst+3
        ENDM
```

```
; duplicate an integer three times
dup32  MACRO  dst0,dst1,dst2,src
       movf   src +0,w
       movwf  dst0+0
       movwf  dst1+0
       movwf  dst2+0
       movf   src +1,w
       movwf  dst0+1
       movwf  dst1+1
       movwf  dst2+1
       movf   src +2,w
       movwf  dst0+2
       movwf  dst1+2
       movwf  dst2+2
       movf   src +3,w
       movwf  dst0+3
       movwf  dst1+3
       movwf  dst2+3
       ENDM

; XOR and add dst += (src0 ^ src1 ^ src2)
xadd32 MACRO  dst,src0,src1,src2
       movf   src0+3,w
       xorwf  src1+3,w
       xorwf  src2+3,w
       addwf  dst +3,f

       movf   src0+2,w
       xorwf  src1+2,w
       xorwf  src2+2,w
       addwfc dst +2,f

       movf   src0+1,w
       xorwf  src1+1,w
       xorwf  src2+1,w
       addwfc dst +1,f

       movf   src0+0,w
       xorwf  src1+0,w
       xorwf  src2+0,w
       addwfc dst +0,f
       ENDM

; XOR and sub dst -= (src0 ^ src1 ^ src2)
xsub32 MACRO  dst,src0,src1,src2
       movf   src0+3,w
       xorwf  src1+3,w
       xorwf  src2+3,w
       subwf  dst +3,f

       movf   src0+2,w
       xorwf  src1+2,w
       xorwf  src2+2,w
       subwfb dst +2,f

       movf   src0+1,w
       xorwf  src1+1,w
       xorwf  src2+1,w
       subwfb dst +1,f

       movf   src0+0,w
       xorwf  src1+0,w
```

```
        xorwf  src2+0,w
        subwfb dst +0,f
        ENDM



;==============================================================================
; Main entrance
;==============================================================================
        org    0x00
testv   movlf32       v0,0x01234567 ; initialize test vector
        movlf32       v1,0x89abcdef
        movlf32       k0,0x00112233
        movlf32       k1,0x44556677
        movlf32       k2,0x8899aabb
        movlf32       k3,0xccddeeff
        nop
        call   encrypt        ; encrypt
        nop                   ; add breakpoint here
        call   decrypt        ; decrypt
        nop                   ; put breakpoint here!
        goto   $              ; freeze microcontroller




;==============================================================================
; Encrypting Routine
;==============================================================================
encrypt
        movlf  cnt,.32              ; Prepare for 32 loops
        clrf32 sum                  ; Reset sum
enc1    addl32 sum,0x9e3779b9       ; add 0x9e3779b9 to summing buffer

        ; Process v0
        dup32  t0,t1,t2,v1    ; t0 = t1 = t2 = v1
        rlf32  t0             ; t0 = (t0 << 4) + k0
        rlf32  t0
        rlf32  t0
        rlf32  t0
        add32  t0,k0

        add32  t1,sum         ; t1 = (t1 + sum)

        rrf32  t2             ; t2 = (t2 >> 5) + k1
        rrf32  t2
        rrf32  t2
        rrf32  t2
        rrf32  t2
        add32  t2,k1

        xadd32 v0,t0,t1,t2    ; v0 += (t0 ^ t1 ^ t2)

        ; Process v1
        dup32  t0,t1,t2,v0    ; t0 = t1 = t2 = v0
        rlf32  t0             ; t0 = (t0 << 4) + k2
        rlf32  t0
        rlf32  t0
        rlf32  t0
        add32  t0,k2

        add32  t1,sum         ; t1 = (t1 + sum)
```

```
       rrf32  t2            ; t2 = (t2 >> 5) + k3
       rrf32  t2
       rrf32  t2
       rrf32  t2
       rrf32  t2

       add32  t2,k3
       xadd32 v1,t0,t1,t2   ; v1 += (t0 ^ t1 ^ t2)

       decfsz cnt,f
       bra    enc1
       return



;===============================================================================
; Decrypting Routine
;===============================================================================
decrypt
       movlf        cnt,.32              ; Prepare for 32 loops
       movlf32      sum,0xc6ef3720       ; sum = 0xc6ef3720
dec1   ; Process v1
       dup32  t0,t1,t2,v0   ; t0 = t1 = t2 = v0
       rlf32  t0            ; t0 = (t0 << 4) + k2
       rlf32  t0
       rlf32  t0
       rlf32  t0
       add32  t0,k2

       add32  t1,sum        ; t1 = (t1 + sum)

       rrf32  t2            ; t2 = (t2 >> 5) + k3
       rrf32  t2
       rrf32  t2
       rrf32  t2
       rrf32  t2
       add32  t2,k3
       xsub32 v1,t0,t1,t2   ; v0 += (t0 ^ t1 ^ t2)

       ; Process v0
       dup32  t0,t1,t2,v1   ; t0 = t1 = t2 = v1
       rlf32  t0            ; t0 = (t0 << 4) + k0
       rlf32  t0
       rlf32  t0
       rlf32  t0
       add32  t0,k0

       add32  t1,sum        ; t1 = (t1 + sum)
       rrf32  t2            ; t2 = (t2 >> 5) + k1
       rrf32  t2
       rrf32  t2
       rrf32  t2
       rrf32  t2
       add32  t2,k1
       xsub32 v0,t0,t1,t2   ; v0 += (t0 ^ t1 ^ t2)

       subl32 sum,0x9e3779b9      ; subtract 0x9e3779b9 from summing buffer

       decfsz cnt,f
       bra    dec1
       return

       END
```

Software Usage

To perform encryption, user can simply put cipher key on k0:k3 and plain text at v0:v1 then call encrypt routine. After encrypting routine has been done, cipher text can be retrieved from v0:v1. In addition cipher key buffer is unaltered during encryption process, so that decryption can be directly performed after cipher text located on v0:v1. Decryption process has the same step as encryption, the difference is that initial data on v0:v1 is cipher Text and resulting data is plain text and the processing function is decrypting routine.

Performance

<u>Test Vector Result</u>

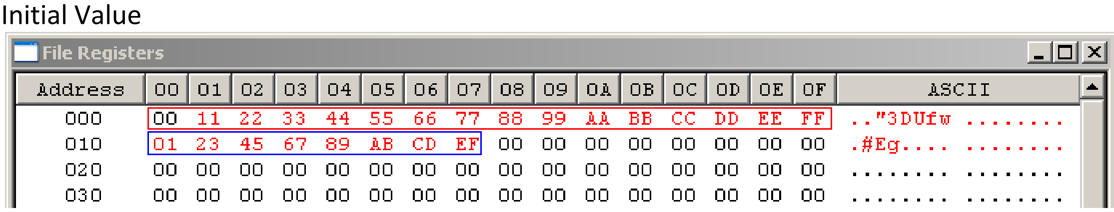
To verify the implementation, author has tested the assembly implementation to process following data:

Ciphering Test Vector		
Plain Text	Cipher key	Cipher Text
0x0123456789abcdef	0x00112233445566778899aabbccddeeff	0x126c6b92c0653a3e

Deciphering Test Vector		
Cipher Text	Cipher Key	Plain Text
0x126c6b92c0653a3e	0x00112233445566778899aabbccddeeff	0x0123456789abcdef

Below are two screenshot showing data progression during encrypting. One marked red are cipher key while one marked blue is data that being processed. At the initial value, the one that marked blue is showing plain text while at the final value showing cipher text.

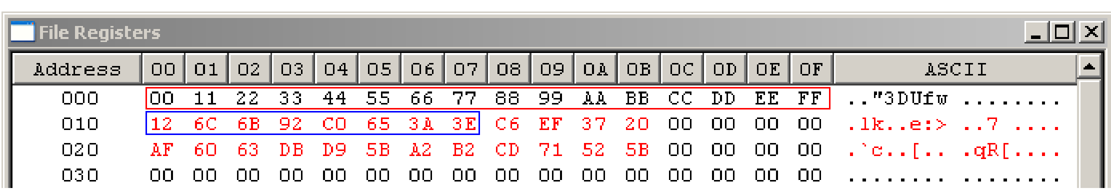
1. Initial Value



File Registers

Address	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	ASCII
000	00	11	22	33	44	55	66	77	88	99	AA	BB	CC	DD	EE	FF	.."3DUfw ....
010	01	23	45	67	89	AB	CD	EF	00	00	00	00	00	00	00	00	.#Eg.... ....
020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
030	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....

2. Final Value



File Registers

Address	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	ASCII
000	00	11	22	33	44	55	66	77	88	99	AA	BB	CC	DD	EE	FF	.."3DUfw ....
010	12	6C	6B	92	C0	65	3A	3E	C6	EF	37	20	00	00	00	00	.lk..e:> ..7 ....
020	AF	60	63	DB	D9	5B	A2	B2	CD	71	52	5B	00	00	00	00	.`c..[.. .qR[....
030	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....

<u>Speed Test Result</u>

The table below is showing speed test result, assuming that 1 cycle is equal to 1 microsecond.

Parameter	Required Number of Cycle	Speed
Key Setup	0	~
Encryption	6826 cycle	585.9 byte/s
Decryption	6830 cycle	585.6 byte/s

Conclusion

Tiny Encryption Algorithm can be implemented in PIC18F4550 with satisfactory result. At 4 MHz (1 microsecond per cycle) working frequency, the implementation shows 586 byte/s speed.

History

Date	Document Version	Code Version	Description
January 19, 2009	1.0	1.0	Initial Release

Reference

Tiny Encryption Algorithm, http://en.wikipedia.org/wiki/Tiny_Encryption_Algorithm

PIC18F4550 datasheet [DS39632D], http://www.microchip.com

Author

Edi Permadi is an Electrical Engineering student of President University. He dedicated his effort to develop hardware based cryptographic device. He is currently doing his final project entitled “PSTN Crypto Phone”, that provides secure communication over telephone line.

He has been doing self research on optimizing the implementation of various cryptographic and hash function. He also just started doing self research that focus on avoiding side channel attack due to cryptographic device.

He is currently working as a part time employee at an International Outsourcing Company headquartered at Singapore as an embedded system developer.