KASUMI Block Cipher on the StarCore SC140 Core

by Mao Zeng

With the rapid growth of wireless services, security in wireless communications has become ever more crucial. Various security algorithms have been developed for wireless systems to provide users with effective and secure communications. The KASUMI block cipher is widely used for security in many synchronous wireless standards. For example, the A5/3 encryption algorithm used in GSM high-level protection against eavesdropping, the GEA3 algorithm adopted by GPRS for data confidentiality, and the $f8/f9$ algorithms specified in 3GPP systems for confidentiality and data integrity are all algorithms based on the 64-bit KASUMI block cipher. The KASUMI is based on a previous block cipher known as MISTY1, which was chosen as the foundation for the 3GPP ciphering algorithm because of its proven security against the most advanced methods for breaking block ciphers, namely cryptanalysis techniques. \textit{KASUMI} is the Japanese word for \textit{misty}. This application note describes how to implement the KASUMI cipher on a Freescale StarCore™-based SC140 DSP.

CONTENTS

1 Basics of the KASUMI Block Cipher ................. 2
2 StarCore Implementation ........................................ 4
  2.1 Code Development .................................................. 4
  2.2 Optimization in C .................................................... 4
  2.3 Optimization in Assembly ...................................... 7
3 Performance Results ............................................... 8
4 References ............................................................... 8
1 Basics of the KASUMI Block Cipher

The KASUMI is a Feistel cipher with eight rounds (see Figure 1). It operates on a 64-bit data block $I$ using a 128-bit key $K$. The 64-bit input string $I$ is divided into two 32-bit strings $L_0$ and $R_0$, where $I = L_0 || R_0$. For each integer $i$ with $1 \leq i \leq 8$, the $i^{th}$ round function of KASUMI is constituted as shown in Equation 1.

$$R_i = L_{i-1}, \quad L_i = R_{i-1} \oplus f_i(L_{i-1}, RK_i)$$

where $f_i$ denotes the round function with $L_{i-1}$ and round key $RK_i$ as inputs. The output result of the KASUMI is equal to the 64-bit string ($L_8$ || $R_8$) offered at the end of the eighth round. The $f_i()$ function takes a 32-bit input and returns a 32-bit output under the control of a round key $RK_i$ where the round key comprises the subkey triplet of ($KL_i$, $KO_i$, $KI_i$). The function itself is constructed from two sub-functions:

- $FL()$. Takes a 32-bit data input and a 32-bit subkey $KL_i$ and it returns a 32-bit output, as shown in Figure 1. The main operations of the $FL$ function are 16-bit AND operations, 16-bit OR operations, and 1-bit left rotation operations.

- $FO()$. Takes a 32-bit data input and two sets of subkeys, a 48-bit subkey $KO_i$ and a 48-bit sub-key $KI_i$, and it generates a 32-bit data output. The $FO$ function comprises three $FI$ functions and six XOR operations.

The $FI$ function takes a 16-bit data input and 16-bit sub-key $KI_{i,j}$. Two S-boxes are $S7$, which maps a 7-bit input to a 7-bit output, and $S9$, which maps a 9-bit input to a 9-bit output, are used in the $FI$ function to provide non-linearity to KASUMI. The details of the $FI$ function and the S-boxes are defined in [2]. The $fi()$ function has two different forms, depending on whether it is an even or odd round. For rounds 1, 3, 5, and 7, it is defined as shown in Equation 2.

$$f_i(I, RK_i) = FO(FL(I, LK_i)KO_iKI_i)$$

For rounds 2, 4, 6, and 8, it is defined as shown in Equation 3.

$$f_i(I, RK_i) = FL(FO(I, KO_iKI_i)LK_i)$$

See Appendix A for a detailed C implementation of the KASUMI cipher.
Figure 1. Components of the KASUMI Block Cipher
2 StarCore Implementation

The StarCore SC140 core is a flexible programmable DSP core that enables the emergence of computationally-intensive communications applications by providing exceptional performance, low power consumption, efficient compatibility, and compact code density. This core efficiently deploys a variable-length execution set (VLES) execution model that achieves maximum parallelism by allowing two address generation and four data arithmetic logic units to execute multiple instructions in a single clock cycle.\(^1\) The SC140 core requires programmers to consider both data-level parallelism (DLP) and instruction-level parallelism (ILP). This section describes the implementation and optimization of the KASUMI cipher on the SC140 core.

2.1 Code Development

Writing functions directly in assembly usually offers the greatest flexibility in optimizing code. However, this method is a very challenging and time-consuming, and it makes debugging the code more difficult. Therefore, our code development and optimization processes are based on a C implementation. The main steps in this implementation process enable us to achieve high-performance code for the SC140 core in a reasonably short time:

1. Port the code to the SC140 core and profile it using the StarCore adaptations and optimization strategies.
2. Transform the algorithm using function-level C optimization techniques.
3. Implement selected functions in assembly for maximum code performance and minimum code size.

2.2 Optimization in C

To optimize the code, we first port the reference 3GPP C code to the SC140 core. 3GPP provides two sets of test vectors for confidentiality algorithm \(f_8\) and integrity algorithm \(f_9\), respectively. We use the test data for \(f_8\) for verification. The profiler information with the –O3 optimization option is listed in Table 1.

Based on the profiler information and the observations on the assembly code generated by the SC140 compiler, several optimization techniques, including function inlining, unique data typing, pipelining, and loop merging, are applied in the C implementation to improve the performance.

2.2.1 Function Inlining

Function inlining improves execution time by eliminating function-call overhead at the expense of larger code size. The KASUMI profiler information indicates that the overhead of a function-call is more than 20 percent for the \(FI\) and \(FL\) functions. Therefore, we inline these two functions to speed up execution.

---

\(^1\) For details, refer to the SC140 DSP Core Reference Manual, which is available at the web site listed on the back cover of this document.
Functions can be inlined in one of three ways:

- Implicitly, allowing the compiler to select the functions to be inlined. This is done in the Enterprise C compiler by setting the `-Og` compiler option.
- Explicitly, using the `#pragma inline` C statement. To inline a function in several files, place the function in a head file and use the static keyword in each file to prevent the linker from generating duplicate global symbols.
- Manually replacing a function call within the body of the function.

We use the first and the third methods for `FL` and `FI`, respectively. Because `FO` calls the `FI` function three times, as illustrated in Figure 1, inlining the `FI` function significantly increases code size. We modify the `FO` function by merging the three `FI` function calls into a DO-loop, as illustrated Example 1, to reduce code size without reducing efficiency.

**Example 1.** Modified C Code for the FO Function

```c
/**************************** Code Before modification ****************************/
/* static u32 FO( u32 in, int index )                             */
/* { */
/*   u16 left, right; */
/* */
/*   // Split the input into two 16-bit words */
/* */
/*   left  = (u16)(in>>16); */
/* */
/*   right = (u16) in; */
/* */
/*   // Now apply the same basic transformation three times */
/* */
/*   left ^= KOi1[index]; */
/*   left  = FI( left, KIi1[index] ); */
/*   left ^= right; */
/* */
/*   right ^= KOi2[index]; */
/*   right  = FI( right, KIi2[index] ); */
/*   right ^= left; */
/* */
/*   left ^= KOi3[index]; */
/*   left  = FI( left, KIi3[index] ); */
/*   left ^= right; */
/* */
/*   in = (((u32)right)<<16)+left; */
/* */
/*   return( in ); */
/* } */
/**************************** Code After modification **************************/

static u32 FO( u32 in, int index )
{
    u16 x, y, temp;
    int i;
    /* Split the input into two 16-bit words */
    x = (u16)(in>>16);
    y = (u16) in;
    /* Now apply the same basic transformation three times */
    ```
for(i=0; i<3; i++)
{
    x ^= KOi[i][index];
    temp = FI( x, Kli[i][index] );
    x = y;
    y ^= temp;
}

in = (((u32)x)<<16)+ y;
return( in );

2.2.2 Data Typing

Using unique data types for the intermediate local variables can prevent the compiler from generating unnecessary data transformation operations, such as sign extension, zero extension, and shift left or right by 16-bit, and so on. Using a 32-bit integer type for intermediate variables can reduce the critical path of computation and thus increase execution speed in some cases.

2.2.3 Pipelining

In the FO function, there are small data dependencies within two adjacent function calls of FI. Software pipelining can be used to implement the three FI function calls in FO, as illustrated in Figure 2. Pipelining allows us to take advantage of instruction-level parallelism of the SC140 core and thereby reduce the number of overall execution cycles.

![Figure 2. Pipelining of FI Function Calls](image)

To assist the C compiler in software pipelining, the initial reference to the C code must be modified to eliminate variable dependencies by introducing more local variables for intermediate computational results. However, too many local variables may cause use of the stack for data passing, which costs extra execution cycles (two cycles for each stack access). Therefore, we take special care when introducing temporary variables. The modified C code for FI is shown in Example 2.

Example 2. Modified C Code for FI

```c
// FI function
nine1  = x >> 7;
seven1 = x & 0x7F;

/* Now run the various operations */
nine1 = S9[nine1] ^ seven1;
seven1 = S7[seven1] ^ (nine1 & 0x7F);
```
seven2 = seven1 ^ L_shr(subkey, 9);
nine2 = nine1 ^ (subkey&0x1FF);

nine2 = S9[nine2] ^ seven2;
seven2 = S7[seven2] ^ (nine2 & 0x7F);
temp = (seven2<<9) + nine2;

2.2.4 Loop Merging

Combining multiple loops into a single loop can reduce the size of the generated code and increase instruction-level parallelism, thus increasing speed. Example 3 shows a section of code in KeySchedule() after loop merging.

Example 3. Loop Merging

/*************** Before loop merging ***********************/
/* k16 = (WORD *)k; */
/* for( n=0; n<8; ++n ) */
/* key[n] = (u16)((k16[n].b8[0]<<8) + (k16[n].b8[1])); */
/* */
/* // Now build the K’[] keys */
/* */
/* for( n=0; n<8; ++n ) */
/* Kprime[n] = (u16)(key[n] ^ C[n]); */
/* */
/***************************************************************************/

/* Now build the K’[] keys */

for( n=0; n<8; ++n )
{
    key[n] = (u16)((k[2*n]<<8) + (k[2*n+1]));
    Kprime[n] = (u16)(key[n] ^ C[n]);
}

2.3 Optimization in Assembly

Assembly-level code optimization can maximize execution speed and increase code density. We used the optimized C code and the following strategies to perform the assembly-level optimization:

- To reduce the data critical path and shorten execution time, use the special SC140 instruction ADDL1A to replace ASLA and ADDA in the table look-up operations.
- Shorten the initialization process by reducing data pointers.
- Use equivalent implementation transformations for data packing operations. For example, use the IMAC instruction to realize (seven<<9) + nine in the FI function.
- Use circular buffers to access data arrays of key[n] and Kprime[n] in sub-key constructions.
- To reduce code size, use the D[0–8] data registers and the R[0–8] address registers as long as possible.

When speed is of utmost concern, you can eliminate the overhead of function calls by inlining the FO functions. Most importantly, the redundant data packing/unpacking operations can also be eliminated after function inlining. Also, you can use hardware loops and loop nesting for efficient loop execution.
3 Performance Results

Table 2 summarizes the performance of the KASUMI cipher on the SC140 core at different optimization levels. The optimized assembly code is provided in Appendix B.

<table>
<thead>
<tr>
<th>Optimization Level</th>
<th>Speed</th>
<th>Size</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>KASUMI</td>
<td>Key Schedule</td>
<td>Code</td>
<td>Data</td>
</tr>
<tr>
<td>Reference C (–O3)</td>
<td>1092</td>
<td>220</td>
<td>1350</td>
<td>1424</td>
</tr>
<tr>
<td>Optimized C (–O3)</td>
<td>576</td>
<td>203</td>
<td>1206</td>
<td>1296</td>
</tr>
<tr>
<td>Assembly (speed and size)</td>
<td>467</td>
<td>112</td>
<td>850</td>
<td>1296</td>
</tr>
<tr>
<td>Assembly (speed)</td>
<td>412</td>
<td>112</td>
<td>1042</td>
<td>1296</td>
</tr>
</tbody>
</table>

4 References


Appendix A:
Reference Code

Header file
/*-----------------------------------------------
 * Kasumi.h
 *-----------------------------------------------*/

typedef unsigned char u8;
typedef unsigned short u16;
typedef unsigned long u32;

void KeySchedule( u8 *key );
void Kasumi( u8 *data );

C Code
/*-----------------------------------------------
 * Kasumi.c
 *-----------------------------------------------
 *
 * A sample implementation of KASUMI, the core algorithm for the
 * 3GPP Confidentiality and Integrity algorithms.
 *
 * This has been coded for clarity, not necessarily for efficiency.
 *
 * This will compile and run correctly on both Intel (little endian)
 * and Sparc (big endian) machines. (Compilers used supported 32-bit ints).
 *
 * Version 1.1 08 May 2000
 *-----------------------------------------------*/

#include "Kasumi.h"

/*------- 16 bit rotate left ------------------------------------------*/
#define ROL16(a,b) (u16)((a<<b)|(a>>(16-b))

/*------- unions: used to remove "endian" issues ------------------------*/
typedef union {
  u32 b32;
  u16 b16[2];
  u8  b8[4];
} DWORD;
typedef union {
  u16 b16;
  u8  b8[2];
} WORD;

/*------- globals: The subkey arrays -----------------------------------*/
static u16 KLi1[8], KLi2[8];
static u16 KOi1[8], KOi2[8], KOi3[8];
static u16 KIi1[8], KIi2[8], KIi3[8];

/*---------------------------------------------------------------------
* FI()
* The FI function (fig 3). It includes the S7 and S9 tables.
* Transforms a 16-bit value.
*---------------------------------------------------------------*/
static u16 FI( u16 in, u16 subkey )
{
    u16 nine, seven;
    static u16 S7[] = {
        54, 50, 62, 56, 22, 34, 94, 96, 38, 6, 63, 93, 2, 18,123, 33,
        55,113, 39,114, 21, 67, 65, 12, 47, 73, 46, 27, 25,111,124, 81,
        53, 9,121, 79, 52, 60, 58, 48,101,127, 40,120,104, 70, 71, 43,
        20,122, 72, 61, 23,109, 13,100, 77, 1, 16, 7, 82, 10,105, 9, 28,
        102, 31, 26, 45, 75, 4, 85, 92, 37, 74, 80, 49, 68, 29,115, 44,
    static u16 S9[] = {
        167,239,161,379,391,334,  9,338, 38,226, 48,358,452,385, 90,397,
        183,253,147,331,415,340, 51,362,306,500,262, 82,216,159,356,177,
        175,241,489, 37,206, 17,  0,333, 44,254,378, 58,143,220, 81,400,
        95, 3,315,245, 54,235,218,405,472,264,172,494,371,290,399, 76,
        165,197,395,121,257,480,423,212,240,  8,462,176,406,507,288,223,
        501,407,249,265, 89,186,221,428,184,16,74,440,196,458,421,350,163,
        232,158,134,354, 13,250,491,142,191, 69,193,425,152,227,366,135,
        344,300,276,437,320,113,278, 11,243, 87,317, 36, 93,496, 27,
        487,446,482, 41, 68,156,457,131,326,403,339, 20, 39,115,442,124,
        475,384,508, 53,112,170,479,151,126,169, 73,268,279,321,168,364,
        363,292, 46,499,393,327,324, 24,456,267,157,460,488,426,309,229,
        439,506,208,271,349,401,434,236, 16,209,359, 52, 56,120,199,277,
        465,416,252,287,246,   6, 83,305,420,345,153,502, 65, 61,244,282,
        173,222,418, 67,386,368,261,104,476,291,195,430,  49, 79,166,330,
        280,383,373,128,382,408,155,495,367,388,274,107,459,417, 62,454,
        132,225,203,316,234, 14,301, 91,503,286,424,211,347,307,140,374,
        35,103,125,427, 19,214,453,146,498,314,444,230,256,329,198,285,
        50,116, 78,410, 10,205,510,171,231,  45,139,467,  29, 86,505, 32,
        72, 26,342,150,313,490,431,238,411,325,149,473, 40,119,174,355,
        185,233,389, 71,448,273,372, 55,110,178,322, 12,469,392,369,190,
        1,109,375,137,181, 88, 75,308,260,484, 98,272,370,275,412,111,
        336,318, 4,504,492,259,304, 77,337,435, 21,357,303,332,483, 18,
        47, 85, 25,497,474,289,100,269,296,478,270,106, 31,104,433, 84,
        414,486,394, 96, 99,154,511,148,413,361,409,255,162,215,302,201,
        266,351,343,144,441,365,108,298,251, 34,182,509,138,210,335,133,
        311,352,328,141,396,346,123,319,450,281,429,228,443,481, 92,404,
        485,422,248,297,  23,213,130,466, 22,217,283, 70,294,360,419,127,
        312,377,  7,468,194,  2,117,295,463,258,224,447,247,187, 80,398,
        284,353,105,390,299,471,470,184, 57,200,348, 63,204,188, 33,451,
        97, 30,310,219, 94,160,129,493, 64,179,263,102,189,207,114,402,
        438,477,387,122,192, 42,381,  5,145,118,180,449,293,323,136,380,
        43, 66, 60,455,341,445,202,432,  8,237, 15,376,436,464, 59,461);
    }
/* The sixteen bit input is split into two unequal halves, *
* nine bits and seven bits - as is the subkey */

nine  = (u16)(in>>7);
seven = (u16)(in&0x7F);

/* Now run the various operations */

nine  = (u16)(S9[nine] ^ seven);
seven = (u16)(S7[seven] ^ (nine & 0x7F));

seven ^= (subkey>>9);
nine ^= (subkey&0x1FF);

nine  = (u16)(S9[nine] ^ seven);
seven = (u16)(S7[seven] ^ (nine & 0x7F));

in = (u16)((seven<<9) + nine);

return( in );

/*---------------------------------------------------------------------
* FO()
* The FO() function.
* Transforms a 32-bit value. Uses <index> to identify the
* appropriate subkeys to use.
*---------------------------------------------------------------------*/

static u32 FO( u32 in, int index )
{
    u16 left, right;

    /* Split the input into two 16-bit words */

    left  = (u16)(in>>16);
    right = (u16) in;

    /* Now apply the same basic transformation three times */

    left ^= KOi1[index];
    left  = FI( left, KIi1[index] );
    left ^= right;

    right ^= KOi2[index];
    right  = FI( right, KIi2[index] );
    right ^= left;

    left ^= KOi3[index];
    left  = FI( left, KIi3[index] );
    left ^= right;

    in = (((u32)right)<<16)+left;

    return( in );
KASUMI Block Cipher on the StarCore SC140 Core, Rev. 0

References

}  
/*-------------------------------------------------------------------------------------*/ 
* FL() 
* The FL() function. 
* Transforms a 32-bit value. Uses <index> to identify the 
* appropriate subkeys to use. 
*-------------------------------------------------------------------------------------*/ 
static u32 FL( u32 in, int index )
{
  u16 l, r, a, b;

  /* split out the left and right halves */
  
  l = (u16)(in>>16);
  r = (u16)(in);

  /* do the FL() operations*/
  
  a  = (u16) (l & KLi1[index]);
  r ^= ROL16(a,1);

  b  = (u16)(r | KLi2[index]);
  l ^= ROL16(b,1);

  /* put the two halves back together */
  
  in = (((u32)l)<<16) + r;

  return( in );
}

/*-------------------------------------------------------------------------------------*/ 
* Kasumi() 
* the Main algorithm (fig 1). Apply the same pair of operations 
* four times. Transforms the 64-bit input. 
*-------------------------------------------------------------------------------------*/ 
void Kasumi( u8 *data )
{
  u32 left, right, temp; 
  DWORD *d;
  int n;

  /* Start by getting the data into two 32-bit words (endian correct) */

  d = (DWORD*)data;
  left  = (((u32)d[0].b8[0])<<24)+(((u32)d[0].b8[1])<<16)
    +(d[0].b8[2]<<8)+(d[0].b8[3]);
  right = (((u32)d[1].b8[0])<<24)+(((u32)d[1].b8[1])<<16)
    +(d[1].b8[2]<<8)+(d[1].b8[3]);
  n = 0;
  do{
    temp = FL( left, n   );
    temp = FO( temp,  n++);
    right ^= temp;
  } while(n < 4);
temp = FO( right, n   );
temp = FL( temp,   n++ );
left ^= temp;
}while( n<=7 );

/* return the correct endian result */
d[0].b8[0] = (u8)(left>>24);d[1].b8[0] = (u8)(right>>24);
d[0].b8[1] = (u8)(left>>16);d[1].b8[1] = (u8)(right>>16);
d[0].b8[2] = (u8)(left>>8);d[1].b8[2] = (u8)(right>>8);
d[0].b8[3] = (u8)(left);d[1].b8[3] = (u8)(right);
}  

/*---------------------------------------------*/
* KeySchedule()
* Build the key schedule. Most *key* operations use 16-bit
* subkeys so we build u16-sized arrays that are *endian* correct.
*---------------------------------------------*/
void KeySchedule( u8 *k )
{
  static u16 C[] = {
    0x0123,0x4567,0x89ab,0xcdef, 0xfedc,0xba98,0x7654,0x3210 },
  u16 key[8], Kprime[8];
  WORD *k16;
  int n;

  /* Start by ensuring the subkeys are endian correct on a 16-bit basis */
  k16 = (WORD *)k;
  for( n=0; n<8; ++n )
    key[n] = (u16)((k16[n].b8[0]<<8) + (k16[n].b8[1]));

  /* Now build the K'[i] keys */
  for( n=0; n<8; ++n )
    Kprime[n] = (u16)(key[n] ^ C[n]);

  /* Finally construct the various sub keys */
  for( n=0; n<8; ++n )
  {
    KLl1[n] = ROL16(key[n],1);
    KLl2[n] = Kprime[(n+2)&0x7];
    KOI1[n] = ROL16(key[(n+1)&0x7],5);
    KOI2[n] = ROL16(key[(n+5)&0x7],8);
    KOI3[n] = ROL16(key[(n+6)&0x7],13);
    KII1[n] = Kprime[(n+4)&0x7];
    KII2[n] = Kprime[(n+3)&0x7];
    KII3[n] = Kprime[(n+7)&0x7];
  }
}  

/*---------------------------------------------*/
* end of k a s u m i . c
*---------------------------------------------*/
Appendix B: Optimized Assembly code

;****************************************************************************
; COPYRIGHT © 2004 FreeScale Semiconductor INC.
; FreeScale Semiconductor
; DSPP, Austin
;****************************************************************************

; FILE NAME: Kasumi.asm
; LANGUAGE (optional): Assembly
; TARGET PROCESSOR: Star*Core 140

;**************************** PURPOSE ***************************************

; DESCRIPTION : Implementation of Kasumi cipher defined by 3GPP TS 35.202

; REFERENCES (optional): None.

;**************************** INPUT AND OUTPUT ******************************

; INPUT: pointer to data --- R0

; OUTPUT: none

; SCRATCH VARIABLES:

; IMPORTED REFERENCES: None.

; EXPORTED REFERENCES: None.

;**************************** RESOURCES *************************************

; REGISTERS USED: d0 – d7, d14, d15, r0 – r12, m0 – m2, n0 – n3

; REGISTERS CHANGED: all above registers except d6, d7, r6, r7.

; CYCLE COUNT:
; Typical = 412

; SIZE: 1042 bytes (code) + 1296 bytes (data)

;**************************** REVISION HISTORY ****************************

<table>
<thead>
<tr>
<th>MM/DD/YYYY</th>
<th>Author</th>
<th>CR Number</th>
<th>Brief Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>07/01/2004</td>
<td>Mao Zeng</td>
<td></td>
<td>created the code – optimized for</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>for speed</td>
</tr>
</tbody>
</table>

;**************************** ASSEMBLY CODE *******************************

SECTIONKasumi_dataLOCAL
SECFLAGS ALLOC,WRITE,NOEXECINSTR
ALIGN 8
SECTYPE PROGBITS

__C TYPE VARIABLE
SIZE __C,16,8
DCW 291,17767,35243,52719,65244,47768,30292,12816 ; offset = 0

__S9 TYPE VARIABLE
SIZE __S9,1024,2
DCW 167,239,161,379,391,334,9,338,38,226,48,358,30292,12816  ; offset = 16
DCW 191,69,193,425,152,227,366,135,344,300,276,242,437,320,113
DCW 278,11,243,87,317,36,93,496,27,487,446,482,41,68,156
DCW 252,287,246,6,83,317,36,93,496,27,487,446,482,41,68,156
DCW 280,383,373,128,382,408,155,495,367,388,274,107,459,417,62
DCW 278,11,243,87,317,36,93,496,27,487,446,482,41,68,156
DCW 167,239,161,379,391,334,9,338,38,226,48,358,30292,12816  ; offset = 0

_KOi TYPE VARIABLE
SIZE _KOi,96,2
DS 96 ; offset = 1040

_KLi TYPE VARIABLE
SIZE _KLi,32,2
DS 32 ; offset = 1136

__S7 TYPE VARIABLE
SIZE __S7,128,1
DCB 54,50,62,56,22,34,94,96,38,6,63,93,2,18,123,33,55,113,39,114,21,67,65,12,47,73,46,27,25,111
; offset = 1168
DCB 124,81,53,9,121,79,52,60,58,48,101,127,40,120,104,70,71,43,20,42,72,61,23,109,13,100,77,1,
16,7,82

KASUMI Block Cipher on the StarCore SC140 Core, Rev. 0


DCB
10,105,98,117,116,76,11,89,106,0,125,118,99,86,69,30,57,126,87,112,51,17,5,95,14,90,84,91,8,35,103

DCB
32,97,28,66,102,31,26,45,75,4,85,92,37,74,80,49,68,29,115,44,64,107,108,24,110,83,36,78,42,19,15

DCB
41,88,119,59,3

ENDSEC

SECTION Kasumi_code LOCAL
SECFLAGS ALLOC,NOWRITE,EXECONSTR
SECTYPE PROGBITS
TextStart_Kasumi

;***********************************************************************
; Function _Kasumi, ; Stack frame size: 0
; Calling Convention: 1
; Parameter data passed in register r0
; Returned value ret_Kasumi_1 FL  optimized out
;***********************************************************************

GLOBAL _Kasumi
ALIGN 16
_KasumiTYPE func OPT_SPEED

SIZE _Kasumi,F_Kasumi_end-_Kasumi,16
[
  tfr  d6,d14            ;save d6,d7
  tfr  d7,d15            ;
  adda #>4,r0,r11        ;r11 = &data[4]
  tfra r0,r10            ;r10 = &data[0]
]
[
  dosetup2 L3
doen2  #4
]
[
  moveu.b  (r10)+,d7     ; data[0]
  moveu.b  (r11)+,d6     ; data[4]
]
[
  asll  #<24,d7           ; data[0]<<24
  asll  #<24,d6           ; data[4]<<24
  moveu.b  (r10)+,d1     ; data[1]
  moveu.b  (r11)+,d2     ; data[5]
]
[
  aslw  d1,d3             ; data[1]<<16
]
aslw d2,d4 ; data[5]<<16
moveu.b (r10)+,d1 ; data[2]
moveu.b (r11)+,d2 ; data[6]
]
[
asll #8,d1 ; data[2]<<16
asll #8,d2 ; data[6]<<16
or d3,d7
or d4,d6
tfra r7,r9 ;save r7
moveu.b (r11),d4 ; data[7]
]
[
or d1,d7
or d2,d6
moveu.b (r10),d3 ; data[3]
movel #_KLi,r7 ; r7 = &KLi
]
[
or d3,d7 ; d7 = left
or d4,d6 ; d6 = right
tfra r6,r8 ; save r6
movel #_KLi+16,r12 ; r12 = &KLi + 8
]
[
tfr d4,d4 ; loop alignment
tfr d5,d5 ; loop alignment
movel #_KOII,r6 ;
]

FALIGN
LOOPSTART2
L3
;;;  inline FL(left, n)

extractu #<16,#<16,d7,d1 ; l = (u16)(in>>16)
zxt.w d7,d3 ; r = (u16)in
moveu.w (r7)+,d4 ; KLi1[index]
movew (r12)+,d5 ; KLi2[index]
]
[
and d1,d4 ; a = i&KLi1[index]
tfra r6,r1 ;
]
[
extractu #<1,#<15,d4,d0 ; for ROL16(a,1)
asll #<1,d4 ; for ROL16(a,1)
]
[
eor d3,d0 ; for r^=ROL16(a,1)
zxt.w d4,d2
adda #<2,r6
]
eor d2, d0          ; d0 = \text{r}^{\text{ROL16(a,1)}}
or d0, d5           ; b = \text{r} | KLi2[index]

extractu #<1, #<15, d5, d4 ; for ROL16(b,1)
asll #<1, d5          ; for ROL16(b,1)

eor d1, d4          ; for 1^{= ROL16(b,1)}
zxt.w d5, d3

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
:: inline FO(temp, n++)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

eor d3, d4          ; d4 = left, d0 = right
move.w #8, n3
moveu.w (r1), d3   ; d3 = KOi[index]

eor d3, d4          ; x^{KOi[index]}
move.l #_S9, r0     ; r0 = S9
adda #16, r1

extractu #<16, #<7, d4, d1 ; nine1 = x >> 7
and #127, d4, d5    ; seven1 = x \& 0x7F
moveu.w (r1)+n3, d2 ; d2 = subkey
doen3 #2

and #511, d2, d4    ; d4 = subkey\&0x1FF
asrr #<9, d2        ; d2 = subkey>>9
move.l d1, r3       ; r3 = nine1
move.l d5, r5       ; r5 = seven1

tfra r0, r4          ; r4 = S9
move.l #_S7, r2

dosetup3 L22A
    addl1a r3, r4    ; \&S9[nine1]

moveu.w (r4), d3    ; d6 = S9[nine1]
adda r2, r5          ; \&S7[seven1]

eor d3, d5          ; nine1=S9[nine1]^seven1
push d6
push d7

eor d5, d4          ; nine2 = nine1^{(subkey\&0x1FF)}
and #127, d5, d6     ; nine1\&0x7F
moveu.b (r5),d1 ; d1 = S7[seven1]
]
[
  eor d1,d6 ; seven1 = S7[seven1]^(nine1&0x7F)
  tfr d0,d4 ; d4 = x = y
  move.l d4,r3 ; r3 = nine2
  tfra r0,r4
]

FALIGN
LOOPSTART3
L22A
[
  eor d6,d2 ; seven2 = seven1^(subkey>>9)
  moveu.w (r1)+n3,d3 ; KOi[index]
]
[
  add11a r3,r4 ; &S9[nine2]
  move.l d2,r5 ; r5 = seven2
]
[
  eor d3,d4 ; x^KOi[index]
  moveu.w (r4),d6 ; S9[nine2]
]
[
  extractu #<16,#<7,d4,d7 ; nine1 = x>>7
  eor d6,d2 ; nine2 = S9[nine2]^seven2
  and #127,d4,d1 ; seven1 = x&0x7F
  adda r2,r5 ; &S7[seven2]
  moveu.w (r1)+n3,d5 ; subkey = KIi[index]
]
[
  and #127,d2,d4 ; nine2&0x7F
  and #511,d5,d3 ; subkey & 0x1FF
  move.l d7,r3 ; r3 = nine1
  moveu.b (r5),d6 ; d6 = S7[seven2]
]
[
  eor d6,d4 ; seven2 = S7[seven2]^(nine2&0x7F)
    tfra r0,r4
    move.w #512,d6
]
[
  imac d6,d4,d2 ; temp = (seven2<<9)+nine2
  move.l d1,r5 ; r5 = seven1
  add11a r3,r4 ; &S9[nine1]
]
[
  eor d2,d0 ; y^=temp
  tfr d5,d2 ; subkey
  moveu.w (r4),d6 ; S9[nine1]
]
[
  eor d6,d1 ; nine1=S9[nine1]^seven1

KASUMI Block Cipher on the StarCore SC140 Core, Rev. 0

Freescale Semiconductor
asrr     #<9,d2            ; subkey >> 9
  tfr     d0,d4             ; x = y
  adda    r2,r5    ; &S7[seven1]
]
[
  eor     d1,d3            ; nine2=nine1^(subkey&0x1FF)
  and     #127,d1,d6       ; nine1&0x7F
  moveu.b (r5),d7         ; S7[seven1]
]
[
  eor     d7,d6            ; seven1 = S7[seven1]^(nine1&0x7F)
  move.l  d3,r3            ; r3 = nine2
  tfra    r0,r4
]
LOOPEND3

eor     d6,d2            ; seven2 = seven1^(subkey>>9)
[
  move.l  d2,r5            ; r5 = seven2
  add11a  r3,r4            ; &S9[nine2]
]
[
  aslw    d4,d4            ; x<<16
  moveu.w (r4),d7         ; S9[nine2]
]
[
  eor     d7,d2            ; nine2 = S9[nine2]^seven2
  adda    r2,r5            ; &S7[seven2]
  move.w  #512,d3
]
[
  and     #127,d2,d5       ; nine2&0x7F
  zxt.l   d4
  moveu.b (r5),d6         ; S7[seven2]
]
[
  eor     d6,d5            ; seven2 =S7[seven2]^(nine2&0x7F)
  pop     d6
  pop     d7
]
[
  imac    d3,d5,d2         ; temp =(seven2<<9)+nine2
  or      d4,d0            ; in = (u32)((x<<16)+y)
]
  eor     d2,d0            ; y^=temp
[
  eor     d0,d6            ; right^=temp
  tfra    r6,r1
]
  zxt.l   d6
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  inline FO(right,n)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
[
  lsrw    d6,d4            ; (u16)(in>>16)
zxt.w  d6,d0  ; (u16)in
move.w  #8,n3  ;
moveu.w (r1),d3  ; d3 = KOi[index]
]
[
eor  d3,d4  ; x^KOi[index]
move.l  #_S9,r0  ; r0 = S9
  adda  #16,r1
]
[
  extractu #<16,#<7,d4,d1  ; ninel = x>>7
  and  #127,d4,d5  ; seven1 = x & 0x7F
moveu.w (r1)+n3,d2  ; d2 = subkey
doen3  #2  ;
]
[
  and  #511,d2,d4  ; d4 = subkey&0x1FF
  asrr  #<9,d2  ; d2 = subkey>>9
move.l  d1,r3  ; r3 = nine1
move.l  d5,r5  ; r5 = seven1
]
[
  tfra  r0,r4  ; r4 = S9
move.l  #_S7,r2
]
[
  dosetup3 L22B
    addl1a  r3,r4  ; &S9[ninel]
]
[
moveu.w (r4),d3  ; d6 = S9[ninel]
  adda  r2,r5  ; &S7[seven1]
]
[
eor  d3,d5  ; ninel=S9[ninel]^seven1
push  d6
push  d7
]
[
eor  d5,d4  ; nine2 = ninel^(subkey&0x1FF)
  and  #127,d5,d6  ; ninel&0x7F
moveu.b (r5),d1  ; d1 = S7[seven1]
]
[
eor  d1,d6  ; seven1 = S7[seven1]^((ninel&0x7F)
tfr  d0,d4  ; d4 = x = y
move.l  d4,r3  ; r3 = nine2
tfra  r0,r4
]

FALIGN
LOOPSTART3
L22B
]
eor      d6,d2             ; seven2 = seven1^(subkey>>9)
moveu.w (r1)+n3,d3        ; K01[index]
]
[
add1la r3,r4             ; &S9[nine2]
move.1 d2,r5             ; r5 = seven2
]
[
eor      d3,d4             ; x^K01[index]
moveu.w (r4),d6           ; S9[nine2]
]
[
extractu #<16,#<7,d4,d7    ; nine1 = x>>7
eor      d6,d2             ; nine2 = S9[nine2]^seven2
and      #127,d4,d1        ; seven1 = x&0x7F
adda     r2,r5             ; &S7[seven2]
moveu.w (r1)+n3,d5        ; subkey = KIi[index]
]
[
and      #127,d2,d4        ; nine2&0x7F
and      #511,d5,d3        ; subkey & 0x1FF
move.1 d7,r3             ; r3 = nine1
moveu.b (r5),d6           ; d6 = S7[seven2]
]
[
eor      d6,d4             ; seven2 = S7[seven2]^(nine2&0x7F)
tfra     r0,r4
move.w   #512,d6
]
[
imac d6,d4,d2             ; temp = (seven2<<9)+nine2
move.1 d1,r5             ; r5 = seven1
add1la   r3,r4             ; &S9[nine1]
]
[
eor      d2,d0             ; y^=temp
tfr      d5,d2             ; subkey
moveu.w (r4),d6           ; S9[nine1]
]
[
eor      d6,d1             ; nine1=S9[nine1]^seven1
asrr     #<9,d2             ; subkey >> 9
tfr      d0,d4             ; x = y
adda     r2,r5             ; &S7[seven1]
]
[
eor      d1,d3             ; nine2=nine1^(subkey&0x1FF)
and      #127,d1,d6        ; nine1&0x7F
moveu.b (r5),d7           ; S7[seven1]
]
[
eor      d7,d6             ; seven1 = S7[seven1]^(nine1&0x7F)
move.1 d3,r3             ; r3 = nine2
tfra     r0,r4
]
LOOPEND3

    eor      d6, d2       ; seven2 = seven1^(subkey>>9)

    [move.l   d2, r5       ; r5 = seven2
       add11a  r3, r4       ; &S9[nine2]
    ]

    [zxt.w    d4, d4       ; d4 = x
       moveu.w (r4), d7     ; S9[nine2]
    ]

    [eor      d7, d2       ; nine2 = S9[nine2]^seven2
       adda    r2, r5       ; &S7[seven2]
       move.w  #512, d3
    ]

    [and      #127, d2, d5 ; nine2&0x7F
       moveu.b (r5), d6     ; S7[seven2]
       moveu.w (r7)+, d1    ; KL11[index]
    ]

    [eor      d6, d5       ; seven2 =S7[seven2]^nine2&0x7F)
       pop      d6
       pop      d7
    ]
    ;; end of inline FO

    [and      d4, d1       ; I & KL11[index]
       adda    #<2, r6       ;
       imac    d3, d5, d2    ; temp = (seven2<<9)+nine2 (FO)
    ]

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; inline FL(temp, n++)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

    [extractu #<1,#<15,d1,d3 ; for ROL16(a,1)
       asll     #<1,d1       ; for ROL16(a,1)
       eor      d0, d2       ; r = y^=temp (FO)
       moveu.w (r12)+, d5    ; KL12[index]
    ]

    [eor      d2, d3       ; for r^ROL16(a,1)
       zxt.w    d1, d1
       eor      d1, d3       ; r^=ROL16(a,1)
       or       d3, d5       ; b = r|KL12[index]
    ]

    [extractu #<1,#<15,d5,d2 ; for ROL16(b,1)
       asll     #<1,d5       ; for ROL16(b,1)
    ]

    [eor      d4, d2       ; for I^ROL16(b,1)
       zxt.w    d5, d4       ; for ROL16(b,1)
    ]
eor  d4,d2          ; l^=ROL16(b,1)
aslw d2,d1          ;
or   d1,d3           ; temp = in = ((u32)1)<<16) + r
eor  d3,d7           ; left^=temp
LOOPEND2

[
asrr  #8,d7
asrr  #8,d6
move.b d7,(r10)-    ; save data[3]
move.b d6,(r11)-    ; save data[7]
]
[
asrr  #8,d7
asrr  #8,d6
move.b d7,(r10)-    ; save data[2]
move.b d6,(r11)-    ; save save[6]
]
[
asrr  #8,d7
asrr  #8,d6
move.b d7,(r10)-    ; save data[1]
move.b d6,(r11)-    ; save data[5]
]
[
tfra  r9,r7
tfra  r8,r6          ;restore r6,r7
]
rtsd
[
tfr  d15,d7
  tfr  d14,d6          ; restore d7,d6
  move.b d7,(r10)      ; save data[0]
  move.b d6,(r11)      ; save data[4]
]

GLOBAL F_Kasumi_end
F_Kasumi_end
FuncEnd_Kasumi

;***********************************************************************
; Function _KeySchedule, ; Stack frame size: 40
; Calling Convention: 1
;
; Parameter k  passed in register r0
;
;***********************************************************************

GLOBAL _KeySchedule
ALIGN 16
_KeyScheduleTYPEfunc OPT_SPEED
SIZE _KeySchedule,F_KeySchedule_end-_KeySchedule,16
adda     #32,sp,r3

doen3    #<4

dosetup3 L18
tfra     r3,sp

adda     #>2,r0,r1      ; r0 = &k[0], r1 = &k[2]
move.l   d6,m0         ; save d6

move.l   d7,m1         ; save d7
move.l   #__C,r3       ; r3 = C

adda     #>-32,sp,r5   ; r5 = Kprime
adda     #>-16,sp,r4   ; r4 = Key

move.2w (r3)+,d0:d1    ; d0:d1 =C[2n]:C[2n+1]

FALIGN
LOOPSTART3
L18
[
  zxt.w  d0,d0
  zxt.w  d1,d1
  moveu.b (r0)+,d2      ; k[4n]
  moveu.b (r1)+,d4      ; k[4n+2]
]
[
  asll    #<8,d2
  asll    #<8,d4
  moveu.b (r0)+,d7      ; k[4n+1]
  moveu.b (r1)+,d5      ; k[4n+3]
]
[
  add     d2,d7,d2      ; d2 = key[2n]
  add     d4,d5,d3      ; d3 = key[2n+1]
  adda    #<2,r0        ; point to next words
  adda    #<2,r1        ;
]
[
  eor     d2,d0          ; key[2n] ^ C[2n]
  eor     d3,d1          ; key[2n+1] ^ C[2n+1]
  move.2w d2:d3,(r4)+    ; save key[2n], key[2n+1]
]
[
  move.2w d0:d1,(r5)+    ; save Kprime[2n], Kprime[2n+1]
  move.2w (r3)+,d0:d1    ; load C[] for next
]
LOOPEND3

adda     #>-16,sp,r0               ; Key
adda     #>-32,sp,r1               ; Kprime
]
[  move.l  #_KLi,r2
   tfra r1,r9
]
[  move.l  #_KOIi,r3
   adda     #4,r1
]
[  move.w  #16,m2
  move.w  #<3,n0
]
[  tfra r0,r8
  move.l  #$000000AA,mctl   ; R0,R1 use module address
]
[  doen3   #<8
   ; for(n=0; n<8; n++)
   dosetup3 L19
]
[  move.w  #<4,n1
  move.w  #<8,n2
]
[  move.w  #-39,n3
  moveu.w (r0)+,d0    ; d0 = key[n]
]

FALIGN

LOOPSTART3

L19
[  asrr    #15,d0
  asl     d0,d2
  moveu.w (r0)+n1,d1           ; d1=key[n+1]
  moveu.w (r1)+,d4             ; d4=Kprime[(n+2)&7]
]
[  or      d2,d0                ; d0 = ROL16(key[n],1)
   extractu #5,#11,d1,d3
   asll     #5,d1
   moveu.w (r0)+,d2            ; d2=Key[(n+5)&7]
   moveu.w (r1)+,d5            ; d5=Kprime[(n+3)&7]
]
[  or      d3,d1                ; d1 = ROL16(key[(n+1)&7],5)
   extractu #8,#8,d2,d0
   asll     #8,d2
   moveu.w (r0)+n0,d3           ; d3=Key[(n+6)&7]
   move.w  d0,(r2)+n2           ; save KLi1[n]
]
or  d0,d2  ; d2 = ROL16(key[(n+5)&7],8)
extractu #13,#3,d3,d1
asll  #13,d3
moveu.w  (r1)+n0,d6  ; d6=Kprime[(n+4)&7]
move.w  d1,(r3)+n2  ; save KOi1[n]
]
[
or  d1,d3  ; d3 = ROL16(key[(n+6)&7],13)
move.w  d4,(r2)  ; save KLi2[n]
move.w  d6,(r3)+n2  ; save KII1[n]
]
[
move.w  d2,(r3)+n2  ; save KOi2[n]
moveu.w  (r1)+n1,d6  ; d6=Kprime[(n+7)&7]
]
[
move.w  d5,(r3)+n2  ; save KII2[n]
moveu.w  (r0)+,d0  ; d0 = key[n]
]
[
move.w  d3,(r3)+n2  ; save KOi3[n]
adda  #-14,r2,r2
]
move.w  d6,(r3)+n3  ; save KII3[n]
LOOPEND3
move.l  #0,mctl
[
adda  #-32,sp,r4  ;
move.l  ml,d7  ;restore d7
]
[
tfra  r4,sp  ;
move.l  m0,d6  ;restore d6
]
rt
GLOBAL F_KeySchedule_end
F_KeySchedule_end
FuncEnd_KeySchedule

TextEnd_Kasumi
ENDSEC
How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations not listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GMBH
Technical Information Center
Schatzbogen 7
81829 München, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Technical Information Center
3-20-1, Minami-Azabu, Minato-ku
Tokyo 106-8573, Japan
0120 191014 or +81-3-3440-3569
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T. Hong Kong
+800 2666 8080

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

Information in this document is provided solely to enable system and software implementers to use Freescale Semiconductor products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits or integrated circuits based on the information in this document.

Freescale Semiconductor reserves the right to make changes without further notice to any products herein. Freescale Semiconductor makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale Semiconductor assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters which may be provided in Freescale Semiconductor data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including “Typicals” must be validated for each customer application by customer’s technical experts. Freescale Semiconductor does not convey any license under its patent rights nor the rights of others. Freescale Semiconductor products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Freescale Semiconductor product could create a situation where personal injury or death may occur. Should Buyer purchase or use Freescale Semiconductor products for any such unintended or unauthorized application, Buyer shall indemnify and hold Freescale Semiconductor and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Freescale Semiconductor was negligent regarding the design or manufacture of the part.

Freescale™ and the Freescale logo are trademarks of Freescale Semiconductor, Inc. StarCore is a trademark of StarCore LLC. All other product or service names are the property of their respective owners.