ETSI/SAGE Specification

Version: 1.0
Date: 23rd December 1999

Specification of the 3GPP Confidentiality and Integrity Algorithms

Document 1: *f8* and *f9* Specification

Blank Page

PREFACE

This specification has been prepared by the 3GPP Task Force, and gives a detailed specification of the 3GPP confidentiality algorithm ***f8***, and the 3GPP integrity algorithm ***f9***.

This document is the first of four, which between them form the entire specification of the 3GPP Confidentiality and Integrity Algorithms:

- Specification of the 3GPP Confidentiality and Integrity Algorithms. Document 1: ***f8*** and ***f9*** Algorithm Specifications.
- Specification of the 3GPP Confidentiality and Integrity Algorithms. Document 2: **KASUMI** Algorithm Specification.
- Specification of the 3GPP Confidentiality and Integrity Algorithms. Document 3: Implementors' Test Data.
- Specification of the 3GPP Confidentiality and Integrity Algorithms. Document 4: Design Conformance Test Data.

The normative part of the specification of the ***f8*** (confidentiality) and ***f9*** (integrity) algorithms is in the main body of this document. The annexes to this document are purely informative. Annex 1 contains illustrations of functional elements of the algorithm, while Annex 2 contains an implementation program listing of the cryptographic algorithm specified in the main body of this document, written in the programming language C.

The normative part of the specification of the block cipher (**KASUMI)** on which they are based is in the main body of Document 2. The annexes of that document, and Documents 3 and 4 above, are purely informative.

Blank Page

TABLE OF CONTENTS

1. OUTLINE OF THE NORMATIVE PART 8

2. INTRODUCTORY INFORMATION 8
   2.1. Introduction 8
   2.2. Notation 8
   2.3. List of Variables 9

3. CONFIDENTIALITY ALGORITHM *f8* 11
   3.1. Introduction 11
   3.2. Inputs and Outputs 11
   3.3. Components and Architecture 11
   3.4. Initialisation 12
   3.5. Keystream Generation 12
   3.6. Encryption/Decryption 13

4. INTEGRITY ALGORITHM *f9* 14
   4.1. Introduction 14
   4.2. Inputs and Outputs 14
   4.3. Components and Architecture 14
   4.4. Initialisation 15
   4.5. Calculation 15

ANNEX 1 Figures of the *f8* and *f9* Algorithms 17

ANNEX 2 Simulation Program Listing 19
   Header file 19
   Function *f8* 19
   Function *f9* 21

REFERENCES

[1] 3rd Generation Partnership Project; Technical Specification Group Services and System Aspects; 3G Security; Security Architecture (3G TS 33.102 version 3.2.0)

[2] 3rd Generation Partnership Project; Technical Specification Group Services and System Aspects; 3G Security; Cryptographic Algorithm Requirements; (3G TS 33.105 version 3.1.0)

[3] Specification of the 3GPP Confidentiality and Integrity Algorithms; Document 1: *f8* and *f9* specifications.

[4] Specification of the 3GPP Confidentiality and Integrity Algorithms; Document 2: KASUMI Specification.

[5] Specification of the 3GPP Confidentiality and Integrity Algorithms; Document 3: Implementors' Test Data.

[6] Specification of the 3GPP Confidentiality and Integrity Algorithms; Document 4: Design Conformance Test Data.

[7] Information technology – Security techniques – Message Authentication Codes (MACs). ISO/IEC 9797-1:1999

NORMATIVE SECTION

This part of the document contains the normative specification of the Confidentiality and Integrity algorithms.

1. OUTLINE OF THE NORMATIVE PART

Section 2 introduces the algorithms and describes the notation used in the subsequent sections.

Section 3 specifies the confidentiality algorithm ***f8***.

Section 4 specifies the integrity algorithm ***f9***.

2. INTRODUCTORY INFORMATION

2.1. Introduction

Within the security architecture of the 3GPP system there are two standardised algorithms: A confidentiality algorithm ***f8***, and an integrity algorithm ***f9***. These algorithms are fully specified here. Each of these algorithms is based on the **KASUMI** algorithm that is specified in a companion document[4]. **KASUMI** is a block cipher that produces a 64-bit output from a 64-bit input under the control of a 128-bit key.

The confidentiality algorithm ***f8*** is a stream cipher that is used to encrypt/decrypt blocks of data under a confidentiality key **CK**. The block of data may be between 1 and 5114 bits long. The algorithm uses **KASUMI** in a form of output-feedback mode as a keystream generator.

The integrity algorithm ***f9*** computes a 32-bit MAC (Message Authentication Code) of a given input message using an integrity key **IK.** The approach adopted uses **KASUMI** in a form of CBC-MAC mode.

2.2. Notation

2.2.1. Radix

We use the prefix **0x** to indicate **hexadecimal** numbers.

2.2.2. Conventions

We use the assignment operator '=', as used in several programming languages. When we write

<*variable*> = <*expression*>

we mean that <*variable*> assumes the value that <*expression*> had before the assignment took place. For instance,

$$x = x + y + 3$$

means

(new value of x) becomes (old value of x) + (old value of y) + 3.

2.2.3. Bit/Byte ordering

All data variables in this specification are presented with the most significant bit (or byte) on the left hand side and the least significant bit (or byte) on the right hand side. Where a variable is broken down into a number of sub-strings, the left most (most significant) sub-string is numbered 0, the next most significant is numbered 1 and so on through to the least significant.

For example an n-bit **MESSAGE** is subdivided into 64-bit substrings $\mathbf{MB_0, MB_1 \ldots MB_i}$ so if we have a message:

0x0123456789ABCDEFFEDCBA98765432l086545381AB594FC28786404C50A37…

we have:

$\mathbf{MB_0}$ = 0x0123456789ABCDEF
$\mathbf{MB_1}$ = 0xFEDCBA9876543210
$\mathbf{MB_2}$ = 0x86545381AB594FC2
$\mathbf{MB_3}$ = 0x8786404C50A37…

In binary this would be:

00000001001000110100010101100111100010011010101111001101111101111111111110…

with $\mathbf{MB_0}$ = 00000001001000110100010101100111100010011010101111001101111011111
$\mathbf{MB_1}$ = 11111110110111001011101010011000011101100101010000110010000100000
$\mathbf{MB_2}$ = 10000110010101000101001110000001101010110101100101001111110000010
$\mathbf{MB_3}$ = 10000111100001100100000001001100010100001010001101111…

2.2.4. List of Symbols

=	The assignment operator.
⊕	The bitwise exclusive-OR operation
||	The concatenation of the two operands.
$KASUMI[x]_k$	The output of the **KASUMI** algorithm applied to input value **x** using the key **k**.
X[i]	The i^{th} bit of the variable **X**. (**X = X[0] || X[1] || X[2] || …..**).
Y_i	The i^{th} block of the variable **Y**. **($\mathbf{Y = Y_0 \| \| Y_1 \| \| Y_2 \| \| \ldots}$)**.

2.3. List of Variables

A, B	are 64-bit registers that are used within the ***f8*** and ***f9*** functions to hold intermediate values.
BEARER	a 5-bit input to the ***f8*** function.
BLKCNT	a 64-bit counter used in the ***f8*** function.
BLOCKS	an integer variable indicating the number of successive applications of **KASUMI** that need to be performed, for both the ***f8*** and ***f9*** functions.

CK	a 128-bit confidentiality key.
COUNT	a 32-bit time variant input to both the ***f8*** and ***f9*** functions.
DIRECTION	a 1-bit input to both the ***f8*** and ***f9*** functions indicating the direction of transmission (uplink or downlink).
FRESH	a 32-bit random input to the ***f9*** function.
IBS	the input bit stream to the ***f8*** function.
IK	a 128-bit integrity key.
KM	a 128-bit constant that is used to modify a key. This is used in both the ***f8*** and ***f9*** functions. (It takes a different value in each function).
KS[i]	is the i^{th} bit of keystream produced by the keystream generator.
KSB_i	is the i^{th} block of keystream produced by the keystream generator. Each block of keystream comprises 64 bits.
LENGTH	is an input to the ***f8*** and ***f9*** functions. It specifies the number of bits in the input bitstream (1-5114).
MAC-I	is the 32-bit message authentication code (MAC) produced by the integrity function ***f9***.
MESSAGE	is the input bitstream of LENGTH bits that is to be processed by the ***f9*** function.
OBS	the output bit streams from the ***f8*** function.
PS	is the input padded string processed by the ***f9*** function.
REGISTER	is a 64-bit value that is used within the ***f8*** function.

3. CONFIDENTIALITY ALGORITHM *f8*

3.1. Introduction

The confidentiality algorithm ***f8*** is a stream cipher that encrypts/decrypts blocks of data between 1 and 5114 bits in length.

3.2. Inputs and Outputs

The inputs to the algorithm are given in table 1, the output in table 2:

Parameter	Size (bits)	Comment
COUNT	32	Frame dependent input COUNT[0]…COUNT[31]
BEARER	5	Bearer identity BEARER[0]…BEARER[4]
DIRECTION	1	Direction of transmission DIRECTION[0]
CK	128	Confidentiality key CK[0]….CK[127]
LENGTH	X18[1]	The number of bits to be encrypted/decrypted (1-5114)
IBS	1-5114	Input bit stream IBS[0]….IBS[LENGTH-1]

Table 1. ***f8*** inputs

Parameter	Size (bits)	Comment
OBS	1-5114	Output bit stream OBS[0]….OBS[LENGTH-1]

Table 2. ***f8*** output

3.3. Components and Architecture

(See fig 1 Annex A)

The keystream generator is based on the block cipher **KASUMI** that is specified in [4]. **KASUMI** is used in a form of output-feedback mode and generates the output keystream in multiples of 64-bits.

The feedback data is modified by static data held in a 64-bit register **A**, and an (incrementing) 64-bit counter **BLKCNT**.

[1] X18 is a parameter whose value is yet to be defined. In the sample C-code we treat LENGTH as a 32-bit integer.

3.4. Initialisation

In this section we define how the keystream generator is initialised with the key variables before the generation of keystream bits.

We set the 64-bit register **A** to **COUNT || BEARER || DIRECTION || 0…0**

(left justified with the right most 26 bits set to 0).

i.e. **A = COUNT[0]…COUNT[31] BEARER[0]…BEARER[4] DIRECTION[0] 0…0**

We set counter **BLKCNT** to zero.

We set the key modifier **KM** to 0x55555555555555555555555555555555

We set $\mathbf{KSB_0}$ to zero.

One operation of **KASUMI** is then applied to the register **A**, using a modified version of the confidentiality key.

$$\mathbf{A = KASUMI[\ A\ ]_{CK \oplus KM}}$$

3.5. Keystream Generation

Once the keystream generator has been initialised in the manner defined in section 3.4, it is ready to be used to generate keystream bits. The plaintext/ciphertext to be encrypted/decrypted consists of **LENGTH** bits (1-5114) whilst the keystream generator produces keystream bits in multiples of 64 bits. Between 0 and 63 of the least significant bits are discarded from the last block depending on the total number of bits required by **LENGTH**.

So let **BLOCKS** be equal to (**LENGTH**/64) rounded up to the nearest integer. (For instance, if **LENGTH** = 128 then **BLOCKS** = 2; if **LENGTH** = 129 then **BLOCKS** = 3.)

To generate each keystream block (**KSB**) we perform the following operation:

For each integer **n** with $1 \leq \mathbf{n} \leq \mathbf{BLOCKS}$ we define:

$$\mathbf{KSB_n = KASUMI[\ A \oplus BLKCNT \oplus KSB_{n-1}]_{CK}}$$

where **BLKCNT = n-1**

The individual bits of the keystream are extracted from $\mathbf{KSB_1}$ to $\mathbf{KSB_{BLOCKS}}$ in turn, most significant bit first, by applying the operation:

For **n** = 1 to **BLOCKS**, and for each integer ***i*** with $0 \leq \boldsymbol{i} \leq 63$ we define:

$$\mathbf{KS[((n-1)*64)+i] = KSB_n[i]}$$

3.6. Encryption/Decryption

Encryption/decryption operations are identical and are performed by the exclusive-OR of the input data (IBS) with the generated keystream (KS).

For each integer ***i*** with $0 \leq i \leq$ **LENGTH-1** we define:

$$\mathbf{OBS}[i] = \mathbf{IBS}[i] \oplus \mathbf{KS}[i]$$

4. INTEGRITY ALGORITHM *f9*

4.1. Introduction

The integrity algorithm ***f9*** computes a Message Authentication Code (MAC) on an input message under an integrity key **IK**. The message may be between 1 and 5114 bits in length.

For ease of implementation the algorithm is based on the same block cipher (**KASUMI**) as is used by the confidentiality algorithm ***f8***.

4.2. Inputs and Outputs

The inputs to the algorithm are given in table 3, the output in table 4:

Parameter	Size (bits)	Comment
COUNT-I	32	Frame dependent input COUNT-I[0]…COUNT-I[31]
FRESH	32	Random number FRESH[0]…FRESH[31]
DIRECTION	1	Direction of transmission DIRECTION[0]
IK	128	Integrity key IK[0]…IK[127]
LENGTH	X19[2]	The number of bits to be 'MAC'd
MESSAGE	LENGTH	Input bit stream

Table 3. ***f9*** inputs

Parameter	Size (bits)	Comment
MAC-I	32	Message authentication code MAC-I[0]…MAC-I[31]

Table 4. ***f9*** output

4.3. Components and Architecture

(See fig 2 Annex A)

The integrity function is based on the block cipher **KASUMI** that is specified in [4]. **KASUMI** is used in a chained mode to generate a 64-bit digest of the message input. Finally the leftmost 32-bits of the digest are taken as the output value **MAC-I**.

[2] X19 is a parameter whose value is yet to be defined. In the sample C-code we treat LENGTH as a 32-bit integer.

4.4. Initialisation

In this section we define how the integrity function is initialised with the key variables before the calculation commences.

We set the working variables: **A = 0**
and **B = 0**

We set the key modifier **KM** to 0xAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

We concatenate **COUNT**, **FRESH**, **MESSAGE** and **DIRECTION**. We then append a single '1' bit, followed by between 0 and 63 '0' bits so that the total length of the resulting string **PS** (padded string) is an integral multiple of 64 bits, i.e.:

PS = COUNT[0]…COUNT[31] FRESH[0]…FRESH[31] MESSAGE[0]… MESSAGE[LENGTH-1] DIRECTION[0] 1 0*

Where 0^* indicates between 0 and 63 '0' bits.

4.5. Calculation

We split the padded string **PS** into 64-bit blocks $\mathbf{PS}_i$ where:

$$\mathbf{PS = PS_0 \,||\, PS_1 \,||\, PS_2 \,||\, \ldots \,||\, PS_{BLOCKS-1}}$$

We perform the following operations for each integer **n** with $0 \leq \mathbf{n} \leq \mathbf{BLOCKS-1}$:

$$\mathbf{A = KASUMI[\, A \oplus PS_n \,]_{IK}}$$
$$\mathbf{B = B \oplus A}$$

Finally we perform one more application of **KASUMI** using a modified form of the integrity key **IK**.

$$\mathbf{B = KASUMI[\, B \,]_{IK \oplus KM}}$$

The 32-bit **MAC-I** comprises the left-most 32 bits of the result.

$$\mathbf{MAC\text{-}I = lefthalf[\, B \,]}$$

i.e. For each integer $\boldsymbol{i}$ with $0 \leq \boldsymbol{i} \leq \mathbf{31}$ we define:

$$\mathbf{MAC\text{-}I[}\boldsymbol{i}\mathbf{] = B[}\boldsymbol{i}\mathbf{].}$$

Bits **B[32]…B[63]** are discarded.

INFORMATIVE SECTION

This part of the document is purely informative and does not form part of the normative specification of KASUMI.

ANNEX 1
Figures of the *f8* and *f9* Algorithms

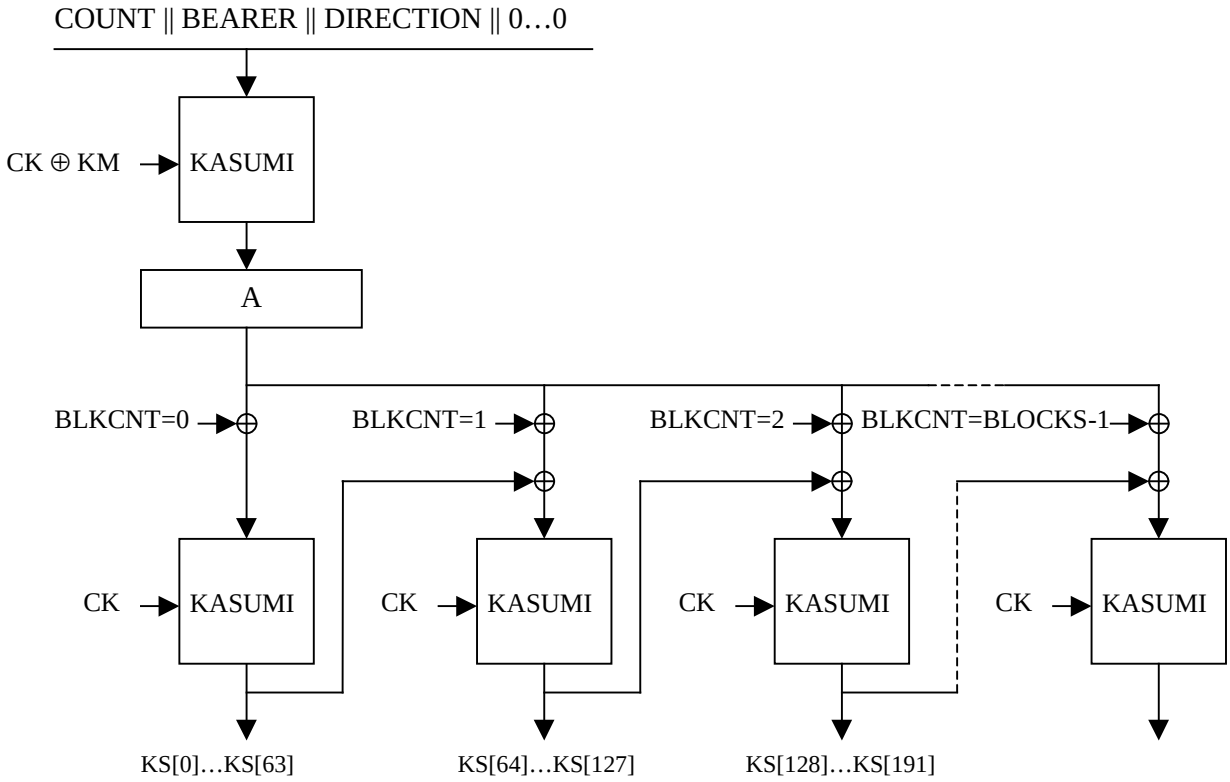

Figure 1: ***f8*** Keystream Generator

Note: **BLKCNT** is specified as a 64-bit counter so there is no ambiguity in the expression $\mathbf{A} \oplus \mathbf{BLKCNT} \oplus \mathbf{KSB}_{n-1}$ where all operands are of the same size. In a practical implementation where the key stream generator is required to produce no more than 5114 bits (80 keystream blocks) only the least significant 7 bits of the counter need to be realised.

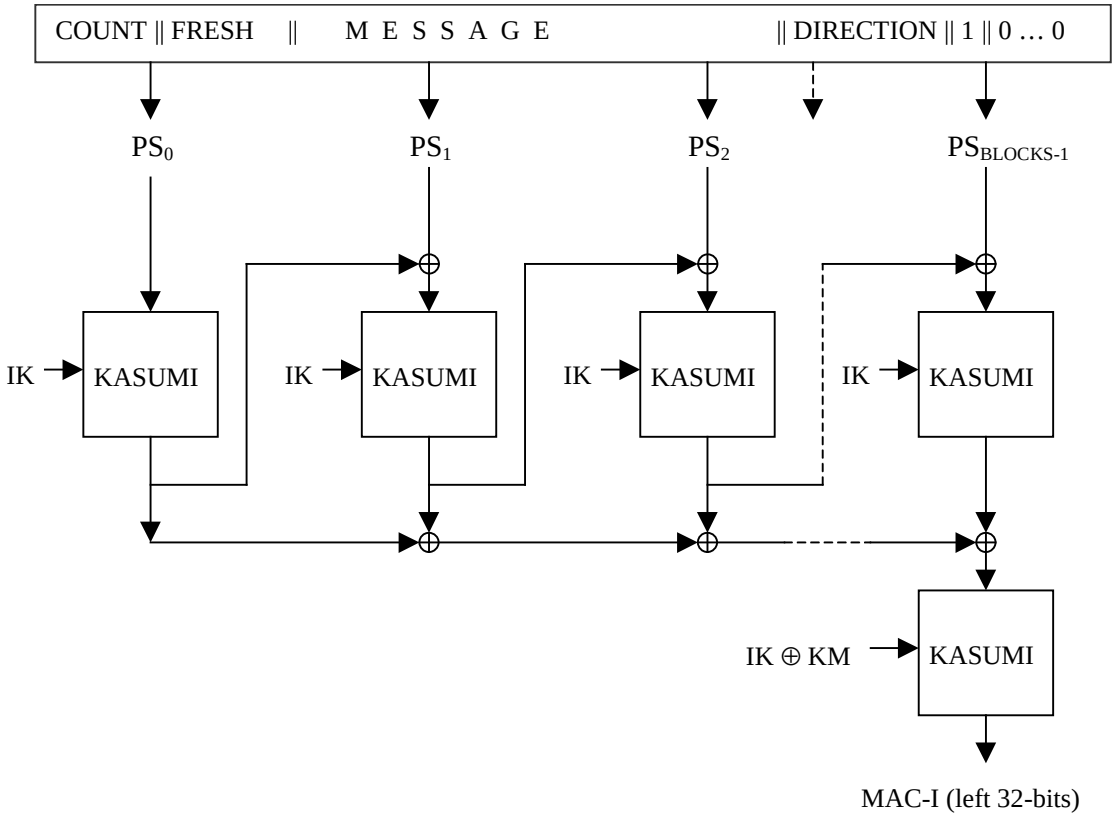

Figure 2 ***f9*** Integrity function

ANNEX 2
Simulation Program Listing

Header file

```
/*---------------------------------------------------------
 *                   Kasumi.h
 *---------------------------------------------------------*/

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

/*----- a 64-bit structure to help with endian issues -----*/

typedef union {
    u32 b32[2];
    u16 b16[4];
    u8  b8[8];
} REGISTER64;

/*------------- prototypes -------------------------------*/

void KeySchedule( u8 *key );
void Kasumi( u8 *data );
u8 * f9( u8 *key,int count,int fresh, int dir,u8 *data,int length );
void f8( u8 *key,int count,int bearer,int dir,u8 *data,int length );
```

Function *f8*

```
/*---------------------------------------------------------
 *              F8 - Confidentiality Algorithm
 *---------------------------------------------------------
 *
 *  A sample implementation of f8, the 3GPP Confidentiality
 *  algorithm.
 *
 *  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.
 *
 *  Version 1.0     05 November  1999
 *
 *---------------------------------------------------------*/

#include "kasumi.h"
#include <stdio.h>

/*---------------------------------------------------------
 * f8()
 *      Given key, count, bearer, direction,  data,
 *      and bit length  encrypt the bit stream
 *---------------------------------------------------------*/
void f8( u8 *key, int count, int bearer, int dir, u8 *data, int length )
{
    REGISTER64 A;        /* the modifier          */
    REGISTER64 temp;     /* The working register  */
    int i, n;
    u8  ModKey[16];      /* Modified key      */
    u16 blkcnt;          /* The block counter */

    /* Start by building our global modifier */

    temp.b32[0]  = temp.b32[1]  = 0;
    A.b32[0]     = A.b32[1]     = 0;
```

```
    /* initialise register in an endian correct manner*/

    A.b8[0]  = (u8) (count>>24);
    A.b8[1]  = (u8) (count>>16);
    A.b8[2]  = (u8) (count>>8);
    A.b8[3]  = (u8) (count);
    A.b8[4]  = (u8) (bearer<<3);
    A.b8[4] |= (u8) (dir<<2);

    /* Construct the modified key and then "kasumi" A */

    for( n=0; n<16; ++n )
        ModKey[n] = (u8)(key[n] ^ 0x55);
    KeySchedule( ModKey );

    Kasumi( A.b8 );  /* First encryption to create modifier */

    /* Final initialisation steps */

    blkcnt = 0;
    KeySchedule( key );

    /* Now run the block cipher */

    while( length > 0 )
    {
        /* First we calculate the next 64-bits of keystream */

        /* XOR in A and BLKCNT to last value */

        temp.b32[0] ^= A.b32[0];
        temp.b32[1] ^= A.b32[1];
        temp.b8[7]  ^= blkcnt;

        /* KASUMI it to produce the next block of keystream */

        Kasumi( temp.b8 );

        /* Set <n> to the number of bytes of input data   *
         * we have to modify.  (=8 if length <= 64)      */

        if( length >= 64 )
            n = 8;
        else
            n = (length+7)/8;

        /* XOR the keystream with the input data stream */

        for( i=0; i<n; ++i )
            *data++ ^= temp.b8[i];
        length -= 64;    /* done another 64 bits  */
        ++blkcnt;        /* increment BLKCNT      */
    }
}

/*-----------------------------------------------------------
 *          e n d    o f    f 8 . c
 *-----------------------------------------------------------*/
```

Function *f9*

```
/*----------------------------------------------------------
 *              F9 - Integrity Algorithm
 *----------------------------------------------------------
 *
 *  A sample implementation of f9, the 3GPP Integrity
 *  algorithm.
 *
 *  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.
 *
 *  Version 1.0     05 November  1999
 *
 *----------------------------------------------------------*/

#include "kasumi.h"
#include <stdio.h>

/*----------------------------------------------------------
 * f9()
 *      Given key, count, fresh, direction, data,
 *      and message length, calculate the hash value
 *----------------------------------------------------------*/
u8 *f9( u8 *key, int count, int fresh, int dir, u8 *data, int length )
{
    REGISTER64 A;    /* Holds the CBC chained data            */
    REGISTER64 B;    /* Holds the XOR of all KASUMI outputs   */
    u8  FinalBit[8] = {0x80, 0x40, 0x20, 0x10, 8,4,2,1};
    u8  ModKey[16];
    static u8 mac_i[4];  /* static memory for the result */
    int i, n;

    /* Start by initialising the block cipher */

    KeySchedule( key );

    /* Next initialise the MAC chain.  Make sure we   *
     * have the data in the right byte order.         *
     * <A> holds our chaining value...                *
     * <B> is the running XOR of all KASUMI o/ps      */

    for( n=0; n<4; ++n )
    {
        A.b8[n]   = (u8)(count>>(24-(n*8)));
        A.b8[n+4] = (u8)(fresh>>(24-(n*8)));
    }
    Kasumi( A.b8 );
    B.b32[0] = A.b32[0];
    B.b32[1] = A.b32[1];

    /* Now run the blocks until we reach the last block */

    while( length >= 64 )
    {
        for( n=0; n<8; ++n )
            A.b8[n] ^= *data++;
        Kasumi( A.b8 );
        length -= 64;
        B.b32[0] ^= A.b32[0];    /* running XOR across */
        B.b32[1] ^= A.b32[1];    /* the block outputs  */
    }

    /* Process whole bytes in the last block */

    n = 0;
    while( length >=8 )
    {
        A.b8[n++] ^= *data++;
        length -= 8;
    }
```

```
    /* Now add the direction bit to the input bit stream  *
     * If length (which holds the # of data bits in the   *
     * last byte) is non-zero we add it in, otherwise     *
     * it has to start a new byte.                        */

    if( length )
    {
        i = *data;
        if( dir )
            i |= FinalBit[length];
    }
    else
        i = dir ? 0x80 : 0;

    A.b8[n++] ^= (u8)i;

    /* Now add in the final '1' bit.  The problem here    *
     * is if the message length happens to be n*64-1.     *
     * If so we need to process this block and then       *
     * create a new input block of 0x8000000000000000.    */

    if( (length==7) && (n==8 ) ) /* then we've filled the block */
    {
        Kasumi( A.b8 );
        B.b32[0] ^= A.b32[0];    /* running XOR accross   */
        B.b32[1] ^= A.b32[1];    /* the block outputs */

        A.b8[0] ^= 0x80;         /* toggle first bit */
        i = 0x80;
        n = 1;
    }
    else
    {
        if( length == 1 )        /* we finished off the last byte */
            i = 0x80;            /* so start a new one.....       */
        else if( length == 0 )   /* we added a new byte of "dir"  */
        {
            A.b8[n-1] ^= 0x40;
            i |= 0x40;
        }
        else
        {
            A.b8[n-1] ^= FinalBit[length+1];
            i |= FinalBit[length+1];
        }
    }


    Kasumi( A.b8 );
    B.b32[0] ^= A.b32[0];    /* running XOR across    */
    B.b32[1] ^= A.b32[1];    /* the block outputs     */

    /* Final step is to KASUMI what we have using the *
     * key XORd with 0xAAAA.....                      */

    for( n=0; n<16; ++n )
        ModKey[n] = (u8)*key++ ^ 0xAA;
    KeySchedule( ModKey );
    Kasumi( B.b8 );

    /* We return the left-most 32-bits of the result */

    for( n=0; n<4; ++n )
        mac_i[n] = B.b8[n];

    return( mac_i );
}

/*---------------------------------------------------------
 *          e n d    o f    f 9 . c
 *---------------------------------------------------------*/
```