

Cryptic

P. Baillehache

May 24, 2020

Contents

1	Interface	1
2	Code	5
2.1	cryptic.c	5
2.2	cryptic-inline.c	20
3	Makefile	26
4	Unit tests	27
5	Unit tests output	37

Introduction

Cryptic library is a C library to cipher and decipher data.

It provides an implementation of the Feistel cipher scheme with the following operating mode: EBC, CBC and CTR.

It uses the PBErr and GSet libraries.

1 Interface

```
// ***** CRYPTIC.H *****  
#ifndef CRYPTIC_H  
#define CRYPTIC_H  
  
// ===== Include =====
```

```

#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include "pberr.h"
#include "gset.h"

// ===== Define =====

#define FUN_CIPHER \
    (*fun)( \
        unsigned char*, \
        unsigned char*, \
        unsigned char*, \
        unsigned long)

#define DEFAULT_OP_MODE FeistelCipheringOpMode_CTR

// ===== Data structures =====

// Operating mode
typedef enum FeistelCipheringOpMode {

    FeistelCipheringOpMode_ECB,
    FeistelCipheringOpMode_CBC,
    FeistelCipheringOpMode_CTR

} FeistelCipheringOpMode;

// Structure for the Feistel cipher
typedef struct FeistelCiphering {

    // GSet of null terminated strings, all the same size,
    // which are the keys to cipher/decipher
    GSetStr* keys;

    // Function to be used during ciphering
    void FUN_CIPHER;

    // Operating mode
    FeistelCipheringOpMode mode;

    // Initialization vector as a null terminated string
    // Used by CBC and CTR operating mode
    // In case of CTR, space for the counter is allocated at the end of
    // the initVector given by the user to append the counter
    unsigned char* initVector;

    // Buffer for the encoding/decoding when using CBC and CTR
    unsigned char* streamBuffer;

    // Counter for the CTR operating mode
    unsigned long counter;

} FeistelCiphering;

// ===== Functions declaration =====

// Static constructor for a Feistel cipher,
// 'keys' is a GSet of null terminated strings, all the same size
// 'fun' is the ciphering function of the form
// void (*fun)(char* src, char* dest, char* key, unsigned long len)
// 'src', 'dest' have same length 'len'

```

```

// 'key' may be of any length
#if BUILDMODE != 0
static inline
#endif
FeistelCiphering FeistelCipheringCreateStatic(
    GSetStr* keys,
    void FUN_CIPHER);

// Function to free the memory used by the static FeistelCiphering
void FeistelCipheringFreeStatic(
    FeistelCiphering* that);

// Function to cipher the message 'msg' with the FeistelCiphering 'that'
// The message length 'lenMsg' must be a multiple of 2
// Return a new string containing the ciphered message
unsigned char* FeistelCipheringCipher(
    FeistelCiphering* that,
    unsigned char* msg,
    unsigned long lenMsg);

// Function to decipher the message 'msg' with the FeistelCiphering
// 'that'
// The message length 'lenMsg' must be a multiple of 2
// Return a new string containing the deciphered message
unsigned char* FeistelCipheringDecipher(
    FeistelCiphering* that,
    unsigned char* msg,
    unsigned long lenMsg);

// Get the operating mode of the FeistelCiphering 'that'
#if BUILDMODE != 0
static inline
#endif
FeistelCipheringOpMode FeistelCipheringGetOpMode(
    const FeistelCiphering* const that);

// Set the operating mode of the FeistelCiphering 'that' to 'mode'
#if BUILDMODE != 0
static inline
#endif
void FeistelCipheringSetOpMode(
    FeistelCiphering* const that,
    FeistelCipheringOpMode mode);

// Get the initialisation vector of the FeistelCiphering 'that'
#if BUILDMODE != 0
static inline
#endif
const unsigned char* FeistelCipheringGetInitVec(
    const FeistelCiphering* const that);

// Set the initialisation vector of the FeistelCiphering 'that'
// to 'initVec'
// Allocate extra memory to append the counter at the end of the
// initialisation vector if the operation mode is CTR
#if BUILDMODE != 0
static inline
#endif
void FeistelCipheringSetInitVec(
    FeistelCiphering* const that,
    const unsigned char* const initVec);

```

```

// Initialise the stream encoding/decoding of the FeistelCiphering 'that'
// with the initialization vector 'initVec'
#if BUILDMODE != 0
static inline
#endif
void FeistelCipheringInitStream(
    FeistelCiphering* const that,
    const unsigned char* const initVec);

// Function to cipher a stream of messages 'msg' with the
// FeistelCiphering 'that'
// The messages length 'lenMsg' must be a multiple of 2
// The messages of the 'streamIn' are consumed one after the other
// and the resulting ciphered messages is appended in the same order
// to 'streamOut'
// Memory used by the messages from the 'streamIn' is freed
// 'lenMsg' must be at least sizeof(that->counter) + 1
void FeistelCipheringCipherStream(
    FeistelCiphering* that,
    GSetStr* const streamIn,
    GSetStr* const streamOut,
    const unsigned long lenMsg);

// Function to decipher a stream of messages 'msg' with the
// FeistelCiphering 'that'
// The messages length 'lenMsg' must be a multiple of 2
// The messages of the 'streamIn' are consumed one after the other
// and the resulting deciphered messages is appended in the same order
// to 'streamOut'
// Memory used by the messages from the 'streamIn' is freed
// 'lenMsg' must be at least sizeof(that->counter) + 1
void FeistelCipheringDecipherStream(
    FeistelCiphering* that,
    GSetStr* const streamIn,
    GSetStr* const streamOut,
    const unsigned long lenMsg);

// Get the required size of the initialisation vector for the
// FeistelCiphering 'that' for messages of length 'lenMsg'
#if BUILDMODE != 0
static inline
#endif
unsigned long FeistelCipheringGetReqSizeInitVec(
    const FeistelCiphering* const that,
    const unsigned long lenMsg);

// Function to cipher a file 'fpIn' with the FeistelCiphering 'that'
// Save the result in the file 'fpOut'.
// Uses block of size equals to the key size for ECB or computed from
// the initialization vector for CBC and CTR.
// Keys must have been set and the stream initialised prior
// to calling this function
void FeistelCipheringCipherFile(
    FeistelCiphering* that,
    FILE* const fpIn,
    FILE* const fpOut);

// Function to decipher a file 'fpIn' with the FeistelCiphering 'that'
// Save the result in the file 'fpOut'.
// Uses block of size equals to the key size for ECB or computed from
// the initialization vector for CBC and CTR.
// Keys must have been set and the stream initialised prior

```

```

// to calling this function
void FeistelCipheringDecipherFile(
    FeistelCiphering* that,
    FILE* const fpIn,
    FILE* const fpOut);

// Get the default size of blocks for the FeistelCiphering 'that'
// It's the key size for ECB or computed from
// the initialization vector for CBC and CTR.
#if BUILDMODE != 0
static inline
#endif
unsigned long FeistelCipheringGetDefaultSizeBlock(
    const FeistelCiphering* const that);

// ===== inliner =====

#if BUILDMODE != 0
#include "cryptic-inline.c"
#endif

#endif

```

2 Code

2.1 cryptic.c

```

// ***** CRYPTIC.C *****

// ===== Include =====
#include "cryptic.h"
#if BUILDMODE == 0
#include "cryptic-inline.c"
#endif

// ===== Functions implementation =====

// Function to free the memory used by the static FeistelCiphering
void FeistelCipheringFreeStatic(
    FeistelCiphering* that) {

#if BUILDMODE == 0

    if (that == NULL) {

        CrypticErr->_type = PBErrTypeNullPointer;
        sprintf(
            CrypticErr->_msg,
            "'that' is null");
        PBErrCatch(CrypticErr);

    }

#endif

// Reset pointers
that->keys = NULL;
that->fun = NULL;

```

```

// Free memory
if (that->initVector != NULL) {

    free(that->initVector);
    that->initVector = NULL;

}

if (that->streamBuffer != NULL) {

    free(that->streamBuffer);
    that->streamBuffer = NULL;

}

}

// Function to cipher the message 'msg' with the FeistelCiphering 'that'
// The message length 'lenMsg' must be a multiple of 2
// Return a new string containing the ciphered message
unsigned char* FeistelCipheringCipher(
    FeistelCiphering* that,
    unsigned char* msg,
    unsigned long lenMsg) {

#ifdef BUILDMODE == 0

    if (that == NULL) {

        CrypticErr->_type = PBErrTypeNullPointer;
        sprintf(
            CrypticErr->_msg,
            "'that' is null");
        PBErrCatch(CrypticErr);

    }

    if (msg == NULL) {

        CrypticErr->_type = PBErrTypeNullPointer;
        sprintf(
            CrypticErr->_msg,
            "'msg' is null");
        PBErrCatch(CrypticErr);

    }

    if ((lenMsg % 2) != 0) {

        CrypticErr->_type = PBErrTypeInvalidArg;
        sprintf(
            CrypticErr->_msg,
            "'lenMsg' is not multiple of 2 (%lu)",
            lenMsg);
        PBErrCatch(CrypticErr);

    }

#endif

    // Allocate memory for the ciphered message

```

```

unsigned char* cipheredMsg =
    PBErrMalloc(
        CrypticErr,
        (lenMsg + 1));

// Initialized the ciphered message with the initial message
memcpy(
    cipheredMsg,
    msg,
    lenMsg + 1);

// Declare a variable to memorize the half length of the message
unsigned long halfLenMsg = lenMsg / 2;

// Allocate memory for the ciphering function
unsigned char* str =
    PBErrMalloc(
        CrypticErr,
        lenMsg);

// Loop on keys
GSetIterForward iter = GSetIterForwardCreateStatic(that->keys);
do {

    // Get the key
    unsigned char* key = GSetIterGet(&iter);

    // Copy right half of the current ciphered message into the left
    // of the temporary string
    memcpy(
        str,
        cipheredMsg + halfLenMsg,
        halfLenMsg);

    // Cipher the right half and store it into the right of the
    // temporary string
    (that->fun)(
        cipheredMsg + halfLenMsg,
        str + halfLenMsg,
        key,
        halfLenMsg);

    // Apply the XOR operator on the half right of the temporary
    // string with the left half of the ciphered message
    for (
        int iChar = halfLenMsg;
        iChar--;) {

        str[halfLenMsg + iChar] =
            str[halfLenMsg + iChar] ^
            cipheredMsg[iChar];

    }

    // Copy the temporary string into the ciphered message
    memcpy(
        cipheredMsg,
        str,
        lenMsg);

} while (GSetIterStep(&iter));

```

```

// Exchange the two halves of the ciphered message
for (
    int iChar = halfLenMsg;
    iChar--;) {

    str[halfLenMsg + iChar] = cipheredMsg[iChar];
    str[iChar] = cipheredMsg[halfLenMsg + iChar];

}

memcpy(
    cipheredMsg,
    str,
    lenMsg);

// Free memory
free(str);

// Return the ciphered message
return cipheredMsg;

}

// Function to decipher the message 'msg' with the FeistelCiphering
// 'that'
// The message length 'lenMsg' must be a multiple of 2
// Return a new string containing the deciphered message
unsigned char* FeistelCipheringDecipher(
    FeistelCiphering* that,
    unsigned char* msg,
    unsigned long lenMsg) {

#ifdef BUILDMODE == 0

    if (that == NULL) {

        CrypticErr->_type = PBErrTypeNullPointer;
        sprintf(
            CrypticErr->_msg,
            "'that' is null");
        PBErrCatch(CrypticErr);

    }

    if (msg == NULL) {

        CrypticErr->_type = PBErrTypeNullPointer;
        sprintf(
            CrypticErr->_msg,
            "'msg' is null");
        PBErrCatch(CrypticErr);

    }

    if ((lenMsg % 2) != 0) {

        CrypticErr->_type = PBErrTypeInvalidArg;
        sprintf(
            CrypticErr->_msg,
            "'lenMsg' is not multiple of 2 (%lu)",
            lenMsg);
        PBErrCatch(CrypticErr);

    }

#endif
}

```



```

    }

#endif

    // Allocate memory for the ciphered message
    unsigned char* cipheredMsg =
        PBErrMalloc(
            CrypticErr,
            lenMsg + 1);

    // Initialized the ciphered message with the initial message
    memcpy(
        cipheredMsg,
        msg,
        lenMsg + 1);

    // Declare a variable to memorize the helf length of the message
    unsigned long halfLenMsg = lenMsg / 2;

    // Allocate memory for the ciphering function
    unsigned char* str =
        PBErrMalloc(
            CrypticErr,
            lenMsg);

    // Loop on keys
    GSetIterBackward iter = GSetIterBackwardCreateStatic(that->keys);
    do {

        // Get the key
        unsigned char* key = GSetIterGet(&iter);

        // Copy right half of the current ciphered message into the left
        // of the temporary string
        memcpy(
            str,
            cipheredMsg + halfLenMsg,
            halfLenMsg);

        // Cipher the right half and store it into the right of the
        // temporary string
        (that->fun)(
            cipheredMsg + halfLenMsg,
            str + halfLenMsg,
            key,
            halfLenMsg);

        // Apply the XOR operator on the half right of the temporary
        // string with the left half of the ciphered message
        for (
            int iChar = halfLenMsg;
            iChar--;) {

            str[halfLenMsg + iChar] =
                str[halfLenMsg + iChar] ^
                cipheredMsg[iChar];

        }

        // Copy the temporary string into the ciphered message
        memcpy(

```

```

        cipheredMsg,
        str,
        lenMsg);

} while (GSetIterStep(&iter));

// Exchange the two halves of the ciphered message
for (
    int iChar = halfLenMsg;
    iChar--;) {

    str[halfLenMsg + iChar] = cipheredMsg[iChar];
    str[iChar] = cipheredMsg[halfLenMsg + iChar];

}

memcpy(
    cipheredMsg,
    str,
    lenMsg);

// Free memory
free(str);

// Return the ciphered message
return cipheredMsg;

}

// Function to cipher a stream of messages 'msg' with the
// FeistelCiphering 'that'
// The messages length 'lenMsg' must be a multiple of 2
// The messages of the 'streamIn' are consumed one after the other
// and the resulting ciphered messages is appended in the same order
// to 'streamOut'
// Memory used by the messages from the 'streamIn' is freed
// 'lenMsg' must be at least sizeof(that->counter) + 1
void FeistelCipheringCipherStream(
    FeistelCiphering* that,
    GSetStr* const streamIn,
    GSetStr* const streamOut,
    const unsigned long lenMsg) {

#ifdef BUILDMODE == 0

    if (that == NULL) {

        CrypticErr->_type = PBErrTypeNullPointer;
        sprintf(
            CrypticErr->_msg,
            "'that' is null");
        PBErrCatch(CrypticErr);

    }

    if (streamIn == NULL) {

        CrypticErr->_type = PBErrTypeNullPointer;
        sprintf(
            CrypticErr->_msg,
            "'streamIn' is null");
        PBErrCatch(CrypticErr);

    }


```

```

}

if (streamOut == NULL) {

    CrypticErr->_type = PBErrTypeNullPointer;
    sprintf(
        CrypticErr->_msg,
        "'streamOut' is null");
    PBErrCatch(CrypticErr);
}

if ((lenMsg % 2) != 0) {

    CrypticErr->_type = PBErrTypeInvalidArg;
    sprintf(
        CrypticErr->_msg,
        "'lenMsg' is not multiple of 2 (%lu)",
        lenMsg);
    PBErrCatch(CrypticErr);
}

if (lenMsg <= sizeof(that->counter)) {

    CrypticErr->_type = PBErrTypeInvalidArg;
    sprintf(
        CrypticErr->_msg,
        "'lenMsg' is too small (%lu > %lu)",
        lenMsg,
        sizeof(that->counter));
    PBErrCatch(CrypticErr);
}

#endif

// Loop on the messages from the streamIn
while (GSetNbElem(streamIn) > 0) {

    // Get the message
    unsigned char* msg = (unsigned char*)GSetPop(streamIn);

    // Declare some working variables
    unsigned char* cipheredMsg = NULL;
    unsigned char* xorArg = NULL;

    // Switch according to the operating mode
    switch (FeistelCipheringGetOpMode(that)) {

        case FeistelCipheringOpMode_ECB:

            // Cipher the message
            cipheredMsg =
                FeistelCipheringCipher(
                    that,
                    msg,
                    lenMsg);

            // Append the ciphered message to the streamOut
            GSetAppend(

```

```

        streamOut,
        (char*)cipheredMsg);

    break;

case FeistelCipheringOpMode_CBC:

    // If there has been a previously ciphered message
    if (that->streamBuffer != NULL) {

        // The argument is the previously ciphered message
        xorArg = that->streamBuffer;

    // Else, this is the first ciphered message
    } else {

        // The argument is the initialisation vector
        xorArg = that->initVector;

    }

    // XOR the current message
    for (
        unsigned long iChar = 0;
        iChar < lenMsg;
        ++iChar) {

        msg[iChar] = msg[iChar] ^ xorArg[iChar];

    }

    // Cipher the message
    cipheredMsg =
        FeistelCipheringCipher(
            that,
            msg,
            lenMsg);

    // Append the ciphered message to the streamOut
    GSetAppend(
        streamOut,
        (char*)cipheredMsg);

    // Free memory
    if (that->streamBuffer != NULL) {

        free(that->streamBuffer);

    }

    // Update the buffer with the last ciphered message
    that->streamBuffer = (unsigned char*)strdup((char*)cipheredMsg);

    break;

case FeistelCipheringOpMode_CTR:

    // Update the counter in the initialization vector
    memcpy(
        that->initVector + lenMsg - sizeof(that->counter),
        (char*)&(that->counter),
        sizeof(that->counter));

```

```

        // Cipher the initialisation vector
        cipheredMsg =
            FeistelCipheringCipher(
                that,
                that->initVector,
                lenMsg);

        // XOR the current message with the ciphered initialisation
        // vector
        for (
            unsigned long iChar = 0;
            iChar < lenMsg;
            ++iChar) {

            cipheredMsg[iChar] =
                cipheredMsg[iChar] ^ msg[iChar];

        }

        // Append the ciphered message to the streamOut
        GSetAppend(
            streamOut,
            (char*)cipheredMsg);

        // Increment the counter
        ++(that->counter);

        break;

    default:
        break;

}

// Free the message
free(msg);

}

}

// Function to decipher a stream of messages 'msg' with the
// FeistelCiphering 'that'
// The messages length 'lenMsg' must be a multiple of 2
// The messages of the 'streamIn' are consumed one after the other
// and the resulting deciphered messages is appended in the same order
// to 'streamOut'
// Memory used by the messages from the 'streamIn' is freed
// 'lenMsg' must be at least sizeof(that->counter) + 1
void FeistelCipheringDecipherStream(
    FeistelCiphering* that,
    GSetStr* const streamIn,
    GSetStr* const streamOut,
    const unsigned long lenMsg) {

#ifdef BUILDMODE == 0

    if (that == NULL) {

        CrypticErr->_type = PBErrTypeNullPointer;
        sprintf(

```

```

        CrypticErr->_msg,
        "'that' is null");
    PBErCatch(CrypticErr);
}

if (streamIn == NULL) {

    CrypticErr->_type = PBErTypeNullPointer;
    sprintf(
        CrypticErr->_msg,
        "'streamIn' is null");
    PBErCatch(CrypticErr);
}

if (streamOut == NULL) {

    CrypticErr->_type = PBErTypeNullPointer;
    sprintf(
        CrypticErr->_msg,
        "'streamOut' is null");
    PBErCatch(CrypticErr);
}

if ((lenMsg % 2) != 0) {

    CrypticErr->_type = PBErTypeInvalidArg;
    sprintf(
        CrypticErr->_msg,
        "'lenMsg' is not multiple of 2 (%lu)",
        lenMsg);
    PBErCatch(CrypticErr);
}

if (lenMsg <= sizeof(that->counter)) {

    CrypticErr->_type = PBErTypeInvalidArg;
    sprintf(
        CrypticErr->_msg,
        "'lenMsg' is too small (%lu > %lu)",
        lenMsg,
        sizeof(that->counter));
    PBErCatch(CrypticErr);
}

#endif

// Loop on the messages from the streamIn
while (GSetNbElem(streamIn) > 0) {

    // Get the message
    unsigned char* msg = (unsigned char*)GSetPop(streamIn);

    // Declare some working variables
    unsigned char* decipheredMsg = NULL;
    unsigned char* xorArg = NULL;

    // Switch according to the operating mode

```

```

switch (FeistelCipheringGetOpMode(that)) {

    case FeistelCipheringOpMode_ECB:

        // Decipher the message
        decipheredMsg =
            FeistelCipheringDecipher(
                that,
                msg,
                lenMsg);

        // Append the deciphered message to the streamOut
        GSetAppend(
            streamOut,
            (char*)decipheredMsg);

        break;

    case FeistelCipheringOpMode_CBC:

        // Decipher the message
        decipheredMsg =
            FeistelCipheringDecipher(
                that,
                msg,
                lenMsg);

        // If there has been a previously ciphered message
        if (that->streamBuffer != NULL) {

            // The argument is the previously ciphered message
            xorArg = that->streamBuffer;

            // Else, this is the first ciphered message
        } else {

            // The argument is the initialisation vector
            xorArg = that->initVector;

        }

        // XOR the current message
        for (
            unsigned long iChar = 0;
            iChar < lenMsg;
            ++iChar) {

            decipheredMsg[iChar] =
                decipheredMsg[iChar] ^ xorArg[iChar];

        }

        // Append the deciphered message to the streamOut
        GSetAppend(
            streamOut,
            (char*)decipheredMsg);

        // Free memory
        if (that->streamBuffer != NULL) {

            free(that->streamBuffer);

```

```

    }

    // Update the buffer with the last deciphered message
    that->streamBuffer = (unsigned char*)strdup((char*)msg);

    break;

case FeistelCipheringOpMode_CTR:

    // Update the counter in the initialization vector
    memcpy(
        that->initVector + lenMsg - sizeof(that->counter),
        (char*)&(that->counter),
        sizeof(that->counter));

    // Cipher the initialisation vector
    decipheredMsg =
        FeistelCipheringCipher(
            that,
            that->initVector,
            lenMsg);

    // XOR the current message with the ciphered initialisation
    // vector
    for (
        unsigned long iChar = 0;
        iChar < lenMsg;
        ++iChar) {

        decipheredMsg[iChar] =
            decipheredMsg[iChar] ^ msg[iChar];

    }

    // Append the ciphered message to the streamOut
    GSetAppend(
        streamOut,
        (char*)decipheredMsg);

    // Increment the counter
    ++(that->counter);

    break;

default:
    break;

}

// Free the message
free(msg);

}

// Function to cipher a file 'fpIn' with the FeistelCiphering 'that'
// Save the result in the file 'fpOut'.
// Uses block of size equals to the key size for ECB or computed from
// the initialization vector for CBC and CTR.
// Keys must have been set and the stream initialised prior
// to calling this function

```



```

void FeistelCipheringCipherFile(
    FeistelCiphering* that,
    FILE* const fpIn,
    FILE* const fpOut) {

    #if BUILDMODE == 0

        if (that == NULL) {

            CrypticErr->_type = PBErrTypeNullPointer;
            sprintf(
                CrypticErr->_msg,
                "'that' is null");
            PBErrCatch(CrypticErr);

        }

        if (fpIn == NULL) {

            CrypticErr->_type = PBErrTypeNullPointer;
            sprintf(
                CrypticErr->_msg,
                "'fpIn' is null");
            PBErrCatch(CrypticErr);

        }

        if (fpOut == NULL) {

            CrypticErr->_type = PBErrTypeNullPointer;
            sprintf(
                CrypticErr->_msg,
                "'fpOut' is null");
            PBErrCatch(CrypticErr);

        }

    #endif

    // Get the size of blocks
    unsigned long sizeBlock = FeistelCipheringGetDefaultSizeBlock(that);

    // Create the stream of blocks
    GSetStr streamIn = GSetStrCreateStatic();

    // Load the file in the set of blocks
    while (!feof(fpIn)) {

        // Allocate memory to read the block
        unsigned char* block =
            PBErrMalloc(
                CrypticErr,
                sizeBlock);

        // Read the block
        unsigned long nbRead =
            fread(
                block,
                1,
                sizeBlock,
                fpIn);
    }
}

```

```

    // If the block is incomplete
    if (nbRead != sizeBlock) {

        // Pad with null character
        memset(
            block + nbRead,
            0,
            sizeBlock - nbRead);

    }

    // Add the block to the stream
    GSetAppend(
        &streamIn,
        (char*)block);

}

// Create the stream of ciphered blocks
GSetStr streamOut = GSetStrCreateStatic();

// Cipher the stream
FeistelCipheringCipherStream(
    that,
    &streamIn,
    &streamOut,
    sizeBlock);

// Save the ciphered stream to the output file
while (GSetNbElem(&streamOut) > 0) {

    // Get the block
    unsigned char* block = (unsigned char*)GSetPop(&streamOut);

    // Save it to the output file
    unsigned long nbWrite =
        fwrite(
            block,
            1,
            sizeBlock,
            fpOut);
    (void)nbWrite;

    // Free memory
    free(block);

}

}

// Function to decipher a file 'fpIn' with the FeistelCiphering 'that'
// Save the result in the file 'fpOut'.
// Uses block of size equals to the key size for ECB or computed from
// the initialization vector for CBC and CTR.
// Keys must have been set and the stream initialised prior
// to calling this function
void FeistelCipheringDecipherFile(
    FeistelCiphering* that,
    FILE* const fpIn,
    FILE* const fpOut) {

#ifdef BUILDMODE == 0

```

```

if (that == NULL) {

    CrypticErr->_type = PBETypeNullPointer;
    sprintf(
        CrypticErr->_msg,
        "'that' is null");
    PBEErrorCatch(CrypticErr);

}

if (fpIn == NULL) {

    CrypticErr->_type = PBETypeNullPointer;
    sprintf(
        CrypticErr->_msg,
        "'fpIn' is null");
    PBEErrorCatch(CrypticErr);

}

if (fpOut == NULL) {

    CrypticErr->_type = PBETypeNullPointer;
    sprintf(
        CrypticErr->_msg,
        "'fpOut' is null");
    PBEErrorCatch(CrypticErr);

}

#endif

// Get the size of blocks
unsigned long sizeBlock = FeistelCipheringGetDefaultSizeBlock(that);

// Create the stream of blocks
GSetStr streamIn = GSetStrCreateStatic();

// Load the file in the set of blocks
while (!feof(fpIn)) {

    // Allocate memory to read the block
    unsigned char* block =
        PBEErrorMalloc(
            CrypticErr,
            sizeBlock);

    // Read the block
    unsigned long nbRead =
        fread(
            block,
            1,
            sizeBlock,
            fpIn);

    // If the block is incomplete
    if (nbRead != sizeBlock) {

        // Pad with null character
        memset(
            block + nbRead,

```

```

        0,
        sizeBlock - nbRead);

    }

    // Add the block to the stream
    GSetAppend(
        &streamIn,
        (char*)block);

}

// Create the stream of ciphered blocks
GSetStr streamOut = GSetStrCreateStatic();

// Decipher the stream
FeistelCipheringDecipherStream(
    that,
    &streamIn,
    &streamOut,
    sizeBlock);

// Save the ciphered stream to the output file
while (GSetNbElem(&streamOut) > 0) {

    // Get the block
    unsigned char* block = (unsigned char*)GSetPop(&streamOut);

    // Save it to the output file
    unsigned long nbWrite =
        fwrite(
            block,
            1,
            sizeBlock,
            fpOut);
    (void)nbWrite;

    // Free memory
    free(block);

}

}

```

2.2 cryptic-inline.c

```

// ***** CRYPTIC-INLINE.C *****

// ===== Functions implementation =====

// Static constructor for a Feistel cipher,
// 'keys' is a GSet of null terminated strings, all the same size
// 'fun' is the ciphering function of the form
// void (*fun)(char* src, char* dest, char* key, unsigned long len)
// 'src', 'dest' have same length 'len'
// 'key' may be of any length
#if BUILDMODE != 0
static inline
#endif

```

```

FeistelCiphering FeistelCipheringCreateStatic(
    GSetStr* keys,
    void FUN_CIPHER) {

#ifdef BUILDMODE == 0

    if (keys == NULL) {

        CrypticErr->_type = PBErrTypeNullPointer;
        sprintf(
            CrypticErr->_msg,
            "'keys' is null");
        PBErrCatch(CrypticErr);

    }

    if (fun == NULL) {

        CrypticErr->_type = PBErrTypeNullPointer;
        sprintf(
            CrypticErr->_msg,
            "'fun' is null");
        PBErrCatch(CrypticErr);

    }

#endif

    // Declare a FeistelCiphering and set the properties
    FeistelCiphering c = {

        .keys = keys,
        .fun = fun,
        .mode = DEFAULT_OP_MODE,
        .initVector = NULL,
        .streamBuffer = NULL,
        .counter = 0

    };

    // Return the FeistelCiphering
    return c;

}

// Get the operating mode of the FeistelCiphering 'that'
#ifdef BUILDMODE != 0
static inline
#endif
FeistelCipheringOpMode FeistelCipheringGetOpMode(
    const FeistelCiphering* const that) {

#ifdef BUILDMODE == 0

    if (that == NULL) {

        CrypticErr->_type = PBErrTypeNullPointer;
        sprintf(
            CrypticErr->_msg,
            "'that' is null");
        PBErrCatch(CrypticErr);

    }

#endif

```

```

    }

#endif

    // Return the operating mode
    return that->mode;

}

// Set the operating mode of the FeistelCiphering 'that' to 'mode'
#if BUILDMODE != 0
static inline
#endif
void FeistelCipheringSetOpMode(
    FeistelCiphering* const that,
    FeistelCipheringOpMode mode) {

#if BUILDMODE == 0

    if (that == NULL) {

        CrypticErr->_type = PBErrTypeNullPointer;
        sprintf(
            CrypticErr->_msg,
            "'that' is null");
        PBErrCatch(CrypticErr);

    }

#endif

    // Set the operating mode
    that->mode = mode;

}

// Get the initialisation vector of the FeistelCiphering 'that'
#if BUILDMODE != 0
static inline
#endif
const unsigned char* FeistelCipheringGetInitVec(
    const FeistelCiphering* const that) {

#if BUILDMODE == 0

    if (that == NULL) {

        CrypticErr->_type = PBErrTypeNullPointer;
        sprintf(
            CrypticErr->_msg,
            "'that' is null");
        PBErrCatch(CrypticErr);

    }

#endif

    // Return the initialising vector
    return that->initVector;

}

```

```

// Set the initialisation vector of the FeistelCiphering 'that'
// to 'initVec'
// Allocate extra memory to append the counter at the end of the
// initialisation vector if the operation mode is CTR
#if BUILDMODE != 0
static inline
#endif
void FeistelCipheringSetInitVec(
    FeistelCiphering* const that,
    const unsigned char* const initVec) {

#if BUILDMODE == 0

    if (that == NULL) {

        CrypticErr->_type = PBErrTypeNullPointer;
        sprintf(
            CrypticErr->_msg,
            "'that' is null");
        PBErrCatch(CrypticErr);

    }

#endif

    // Free memory if necessary
    if (that->initVector != NULL) {

        free(that->initVector);

    }

    switch (FeistelCipheringGetOpMode(that)) {

        case FeistelCipheringOpMode_CBC:

            // Copy the initialising vector
            that->initVector = (unsigned char*)strdup((char*)initVec);
            break;

        case FeistelCipheringOpMode_CTR:

            // Allocate memory
            that->initVector =
                (unsigned char*)malloc(
                    strlen((char*)initVec) + 1 + sizeof(that->counter));

            // Init all the bytes to null
            memset(
                that->initVector,
                0,
                strlen((char*)initVec) + 1 + sizeof(that->counter));

            // Copy the initialising vector
            memcpy(
                that->initVector,
                initVec,
                strlen((char*)initVec));

            break;

        default:

```

```

        break;
    }
}

// Initialise the stream encoding/decoding of the FeistelCipherring 'that'
// with the initialization vector 'initVec'
#if BUILDMODE != 0
static inline
#endif
void FeistelCipherringInitStream(
    FeistelCipherring* const that,
    const unsigned char* const initVec) {

#if BUILDMODE == 0

    if (that == NULL) {

        CrypticErr->_type = PBErrTypeNullPointer;
        sprintf(
            CrypticErr->_msg,
            "'that' is null");
        PBErrCatch(CrypticErr);

    }

#endif

    // Initialise the properties used to encode/decode the stream
    that->counter = 0;
    FeistelCipherringSetInitVec(
        that,
        initVec);
    if (that->streamBuffer != NULL) {

        free(that->streamBuffer);

    }

    that->streamBuffer = NULL;

}

// Get the required size of the initialisation vector for the
// FeistelCipherring 'that' for messages of length 'lenMsg'
#if BUILDMODE != 0
static inline
#endif
unsigned long FeistelCipherringGetReqSizeInitVec(
    const FeistelCipherring* const that,
    const unsigned long lenMsg) {

#if BUILDMODE == 0

    if (that == NULL) {

        CrypticErr->_type = PBErrTypeNullPointer;
        sprintf(
            CrypticErr->_msg,
            "'that' is null");

    }

#endif

```



```

        PBErCatch(CrypticErr);
    }

#endif

    // Declare a variable to memorize the size
    unsigned long size = 0;

    // Return the size of initialising vector
    switch (FeistelCipheringGetOpMode(that)) {

        case FeistelCipheringOpMode_CBC:

            size = lenMsg;
            break;

        case FeistelCipheringOpMode_CTR:

            size = lenMsg - sizeof(that->counter);
            break;

        default:

            size = 0;

    }

    // Return the size of initialising vector
    return size;
}

// Get the default size of blocks for the FeistelCiphering 'that'
// It's the key size for ECB or computed from
// the initialization vector for CBC and CTR.
#if BUILDMODE != 0
static inline
#endif
unsigned long FeistelCipheringGetDefaultSizeBlock(
    const FeistelCiphering* const that) {

#if BUILDMODE == 0

    if (that == NULL) {

        CrypticErr->_type = PBErTypeNullPointer;
        sprintf(
            CrypticErr->_msg,
            "'that' is null");
        PBErCatch(CrypticErr);

    }

#endif

#endif

    // Calculate the size of blocks
    unsigned long sizeBlock = 0;
    unsigned char* key = NULL;
    switch (FeistelCipheringGetOpMode(that)) {

        case FeistelCipheringOpMode_ECB:

```

```

        key = (unsigned char*)
            GSetGet(
                that->keys,
                0);
        sizeBlock = strlen((char*)key);
        break;

    case FeistelCipherringOpMode_CBC:
        sizeBlock = strlen((char*)(that->initVector));
        break;

    case FeistelCipherringOpMode_CTR:
        sizeBlock =
            strlen((char*)(that->initVector)) + sizeof(that->counter);
        break;

    default:
        break;
}

// Return the size of block
return sizeBlock;
}

```

3 Makefile

```

# Build mode
# 0: development (max safety, no optimisation)
# 1: release (min safety, optimisation)
# 2: fast and furious (no safety, optimisation)
BUILD_MODE?=0

all: pbmake_wget main

# Automatic installation of the repository PBMake in the parent folder
pbmake_wget:
if [ ! -d ../PBMake ]; then wget https://github.com/BayashiPascal/PBMake/archive/master.zip; unzip master.zip; rm -f

# Check code style
style:
cbo *.h *.c

# Makefile definitions
MAKEFILE_INC=../PBMake/Makefile.inc
include $(MAKEFILE_INC)

# Rules to make the executable
repo=cryptic
${$(repo)_EXENAME}: \
${$(repo)_EXENAME}.o \
${$(repo)_EXE_DEP} \
${$(repo)_DEP}
$(COMPILER) 'echo "${$(repo)_EXE_DEP} ${$(repo)_EXENAME}.o" | tr ' ' '\n' | sort -u' $(LINK_ARG) ${$(repo)_LINK_ARG}

${$(repo)_EXENAME}.o: \
${$(repo)_DIR}/${$(repo)_EXENAME}.c \
${$(repo)_INC_H_EXE} \

```

```

$$($(repo)_EXE_DEP)
$(COMPILER) $(BUILD_ARG) $$($(repo)_BUILD_ARG) 'echo "$$(repo)_INC_DIR" | tr ' ' '\n' | sort -u' -c $$($(repo)_DIR)/

```

4 Unit tests

```

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "cryptic.h"

void CipherringFun(
    unsigned char* src,
    unsigned char* dest,
    unsigned char* key,
    unsigned long len) {

    unsigned long lenKey = strlen((char*)key);
    for (
        unsigned int iChar = 0;
        iChar < len;
        ++iChar) {

        dest[iChar] = src[iChar] + key[iChar % lenKey];

    }

}

void UnitTestFeistelCipherring() {

    GSetStr keys = GSetStrCreateStatic();
    unsigned char keyA[] = "123456";
    unsigned char keyB[] = "abcdef";
    GSetAppend(
        &keys,
        (char*)keyA);
    GSetAppend(
        &keys,
        (char*)keyB);
    unsigned char msg[] = "Hello World.";
    printf("Message:      ");
    for (
        unsigned int iChar = 0;
        iChar < strlen((char*)msg);
        ++iChar) {

        printf(
            "%03u,",
            msg[iChar]);

    }

    printf("\n");
    FeistelCipherring cipher =
        FeistelCipherringCreateStatic(
            &keys,
            &CipherringFun);
    unsigned char* cipheredMsg =

```

```

    FeistelCipheringCipher(
        &cipher,
        msg,
        strlen((char*)msg));
printf("Ciphered message: ");
for (
    unsigned int iChar = 0;
    iChar < strlen((char*)msg);
    ++iChar) {

    printf(
        "%03u,",
        cipheredMsg[iChar]);

}

printf("\n");
unsigned char* decipheredMsg =
    FeistelCipheringDecipher(
        &cipher,
        cipheredMsg,
        strlen((char*)msg));
int ret =
    strcmp(
        (char*)msg,
        (char*)decipheredMsg);
if (ret != 0) {

    CrypticErr->_type = PBErrTypeUnitTestFailed;
    sprintf(
        CrypticErr->_msg,
        "FeistelCipheringCipher/FeistelCipheringDecipher NOK");
    PBErrCatch(CrypticErr);

}

printf(
    "%s\n",
    decipheredMsg);

FeistelCipheringFreeStatic(&cipher);
GSetFlush(&keys);
free(cipheredMsg);
free(decipheredMsg);
printf("UnitTestFeistelCiphering OK\n");

}

void UnitTestFeistelStreamCipheringECB() {

    GSetStr keys = GSetStrCreateStatic();
    unsigned char keyA[] = "123456";
    unsigned char keyB[] = "abcdef";
    GSetAppend(
        &keys,
        (char*)keyA);
    GSetAppend(
        &keys,
        (char*)keyB);
    unsigned char* initVector = (unsigned char*)"!#$%&'()*~=.,";
    GSetStr streamIn = GSetStrCreateStatic();
    unsigned char* msg[2] = {

```

```

        (unsigned char*)"Hello World.    ",
        (unsigned char*)"What's up there?"

    };
    unsigned long lenMsg = strlen((char*)(msg[0]));
    for (int iMsg = 0; iMsg < 2; ++iMsg) {

        GSetAppend(
            &streamIn,
            strdup((char*)(msg[iMsg])));
        printf("Message:      ");
        for (
            unsigned int iChar = 0;
            iChar < lenMsg;
            ++iChar) {

            printf(
                "%03u,",
                msg[iMsg][iChar]);

        }

        printf("\n");
    }

    FeistelCiphering cipher =
        FeistelCipheringCreateStatic(
            &keys,
            &CipheringFun);
    GSetStr streamOut = GSetStrCreateStatic();
    GSetStr streamDecipher = GSetStrCreateStatic();
    FeistelCipheringInitStream(
        &cipher,
        initVector);
    FeistelCipheringSetInitVec(
        &cipher,
        initVector);
    FeistelCipheringCipherStream(
        &cipher,
        &streamIn,
        &streamOut,
        lenMsg);
    while (GSetNbElem(&streamOut) > 0) {

        unsigned char* cipheredMsg = (unsigned char*)GSetPop(&streamOut);
        printf("Ciphered message:  ");
        for (
            unsigned int iChar = 0;
            iChar < lenMsg;
            ++iChar) {

            printf(
                "%03u,",
                cipheredMsg[iChar]);

        }

        printf("\n");
        GSetAppend(
            &streamDecipher,

```

```

        (char*)cipheredMsg);
    }

    FeistelCipheringInitStream(
        &cipher,
        initVector);
    FeistelCipheringDecipherStream(
        &cipher,
        &streamDecipher,
        &streamIn,
        lenMsg);

    int iMsg = 0;
    while (GSetNbElem(&streamIn) > 0) {

        unsigned char* decipheredMsg = (unsigned char*)GSetPop(&streamIn);
        printf("Deciphered message: ");
        for (
            unsigned int iChar = 0;
            iChar < lenMsg;
            ++iChar) {

            printf(
                "%03u,",
                decipheredMsg[iChar]);

        }

        printf("\n");
        printf(
            "%s\n",
            (char*)decipheredMsg);

        int ret =
            strcmp(
                (char*)(msg[iMsg]),
                (char*)decipheredMsg);
        if (ret != 0) {

            CrypticErr->_type = PBErrTypeUnitTestFailed;
            sprintf(
                CrypticErr->_msg,
                "FeistelCipheringCipherECB/FeistelCipheringDecipherECB NOK");
            PBErrCatch(CrypticErr);

        }
        ++iMsg;

        free(decipheredMsg);
    }

    FeistelCipheringFreeStatic(&cipher);
    GSetFlush(&keys);
    printf("UnitTestFeistelStreamCipheringECB OK\n");
}

void UnitTestFeistelStreamCipheringCBC() {

    GSetStr keys = GSetStrCreateStatic();

```

```

unsigned char keyA[] = "123456";
unsigned char keyB[] = "abcdef";
GSetAppend(
    &keys,
    (char*)keyA);
GSetAppend(
    &keys,
    (char*)keyB);
unsigned char* initVector = (unsigned char*)"!#$%&'()*~=.1234";
GSetStr streamIn = GSetStrCreateStatic();
unsigned char* msg[2] = {

    (unsigned char*)"Hello World.    ",
    (unsigned char*)"What's up there?"

};
unsigned long lenMsg = strlen((char*)(msg[0]));
for (int iMsg = 0; iMsg < 2; ++iMsg) {

    GSetAppend(
        &streamIn,
        strdup((char*)(msg[iMsg])));
    printf("Message:          ");
    for (
        unsigned int iChar = 0;
        iChar < lenMsg;
        ++iChar) {

        printf(
            "%03u,",
            msg[iMsg][iChar]);

    }

    printf("\n");

}

FeistelCiphering cipher =
    FeistelCipheringCreateStatic(
        &keys,
        &CipheringFun);
FeistelCipheringSetOpMode(
    &cipher,
    FeistelCipheringOpMode_CBC);
unsigned long reqSize =
    FeistelCipheringGetReqSizeInitVec(
        &cipher,
        lenMsg);
printf(
    "Required initialisation vector's size: %lu\n",
    reqSize);
FeistelCipheringSetInitVec(
    &cipher,
    initVector);
GSetStr streamOut = GSetStrCreateStatic();
GSetStr streamDecipher = GSetStrCreateStatic();
FeistelCipheringInitStream(
    &cipher,
    initVector);
FeistelCipheringCipherStream(
    &cipher,

```

```

    &streamIn,
    &streamOut,
    lenMsg);
while (GSetNbElem(&streamOut) > 0) {

    unsigned char* cipheredMsg = (unsigned char*)GSetPop(&streamOut);
    printf("Ciphered message:  ");
    for (
        unsigned int iChar = 0;
        iChar < lenMsg;
        ++iChar) {

        printf(
            "%03u,",
            cipheredMsg[iChar]);

    }

    printf("\n");
    GSetAppend(
        &streamDecipher,
        (char*)cipheredMsg);
}

FeistelCipheringInitStream(
    &cipher,
    initVector);
FeistelCipheringDecipherStream(
    &cipher,
    &streamDecipher,
    &streamIn,
    lenMsg);

unsigned int iMsg = 0;
while (GSetNbElem(&streamIn) > 0) {

    unsigned char* decipheredMsg = (unsigned char*)GSetPop(&streamIn);
    printf("Deciphered message: ");
    for (
        unsigned int iChar = 0;
        iChar < lenMsg;
        ++iChar) {

        printf(
            "%03u,",
            decipheredMsg[iChar]);

    }

    printf("\n");
    printf(
        "%s\n",
        (char*)decipheredMsg);

    int ret =
        strcmp(
            (char*)(msg[iMsg]),
            (char*)decipheredMsg);
    if (ret != 0) {

        CrypticErr->_type = PBErrTypeUnitTestFailed;

```



```

        sprintf(
            CrypticErr->_msg,
            "FeistelCipheringCipherCBC/FeistelCipheringDecipherCBC NOK");
        PBErCatch(CrypticErr);

    }
    ++iMsg;

    free(decipheredMsg);

}

FeistelCipheringFreeStatic(&cipher);
GSetFlush(&keys);
printf("UnitTestFeistelStreamCipheringCBC OK\n");
}

void UnitTestFeistelStreamCipheringCTR() {

    GSetStr keys = GSetStrCreateStatic();
    unsigned char keyA[] = "123456";
    unsigned char keyB[] = "abcdef";
    GSetAppend(
        &keys,
        (char*)keyA);
    GSetAppend(
        &keys,
        (char*)keyB);
    unsigned char* initVector = (unsigned char*)"!#$%&'(";
    GSetStr streamIn = GSetStrCreateStatic();
    unsigned char* msg[2] = {

        (unsigned char*)"Hello World.  ",
        (unsigned char*)"What's up there?"

    };

    unsigned long lenMsg = strlen((char*)(msg[0]));
    for (int iMsg = 0; iMsg < 2; ++iMsg) {

        GSetAppend(
            &streamIn,
            strdup((char*)(msg[iMsg])));
        printf("Message:      ");
        for (
            unsigned int iChar = 0;
            iChar < lenMsg;
            ++iChar) {

            printf(
                "%03u,",
                msg[iMsg][iChar]);

        }

        printf("\n");

    }

    FeistelCiphering cipher =
        FeistelCipheringCreateStatic(
            &keys,

```

```

        &CipheringFun);
FeistelCipheringSetOpMode(
    &cipher,
    FeistelCipheringOpMode_CTR);
unsigned long reqSize =
    FeistelCipheringGetReqSizeInitVec(
        &cipher,
        lenMsg);
printf(
    "Required initialisation vector's size: %lu\n",
    reqSize);
FeistelCipheringSetInitVec(
    &cipher,
    initVector);
GSetStr streamOut = GSetStrCreateStatic();
GSetStr streamDecipher = GSetStrCreateStatic();
FeistelCipheringInitStream(
    &cipher,
    initVector);
FeistelCipheringCipherStream(
    &cipher,
    &streamIn,
    &streamOut,
    lenMsg);
while (GSetNbElem(&streamOut) > 0) {

    unsigned char* cipheredMsg = (unsigned char*)GSetPop(&streamOut);
    printf("Ciphered message: ");
    for (
        unsigned int iChar = 0;
        iChar < lenMsg;
        ++iChar) {

        printf(
            "%03u,",
            cipheredMsg[iChar]);

    }

    printf("\n");
    GSetAppend(
        &streamDecipher,
        (char*)cipheredMsg);
}

FeistelCipheringInitStream(
    &cipher,
    initVector);
FeistelCipheringDecipherStream(
    &cipher,
    &streamDecipher,
    &streamIn,
    lenMsg);

unsigned int iMsg = 0;
while (GSetNbElem(&streamIn) > 0) {

    unsigned char* decipheredMsg = (unsigned char*)GSetPop(&streamIn);
    printf("Deciphered message: ");
    for (
        unsigned int iChar = 0;

```

```

        iChar < lenMsg;
        ++iChar) {

    printf(
        "%03u,",
        decipheredMsg[iChar]);

}

printf("\n");
printf(
    "%s\n",
    (char*)decipheredMsg);

int ret =
    strcmp(
        (char*)(msg[iMsg]),
        (char*)decipheredMsg);
if (ret != 0) {

    CrypticErr->_type = PBErrTypeUnitTestFailed;
    sprintf(
        CrypticErr->_msg,
        "FeistelCipheringCipherCTR/FeistelCipheringDecipherCTR NOK");
    PBErrCatch(CrypticErr);

}
++iMsg;

free(decipheredMsg);

}

FeistelCipheringFreeStatic(&cipher);
GSetFlush(&keys);
printf("UnitTestFeistelStreamCipheringCTR OK\n");

}

void UnitTestFeistelStreamCipheringFile() {

    GSetStr keys = GSetStrCreateStatic();
    unsigned char keyA[] = "123456";
    unsigned char keyB[] = "abcdef";
    GSetAppend(
        &keys,
        (char*)keyA);
    GSetAppend(
        &keys,
        (char*)keyB);
    FeistelCiphering cipher =
        FeistelCipheringCreateStatic(
            &keys,
            &CipheringFun);

    FILE* fpIn =
        fopen(
            "./cryptic.c",
            "r");
    FILE* fpOut =
        fopen(
            "./cryptic.ciphered",

```

```

        "w");

    unsigned char* initVector = (unsigned char*)"!#$%&'(";
    FeistelCipheringInitStream(
        &cipher,
        initVector);

    FeistelCipheringCipherFile(
        &cipher,
        fpIn,
        fpOut);

    fclose(fpIn);
    fclose(fpOut);

    fpIn =
        fopen(
            "./cryptic.ciphered",
            "r");
    fpOut =
        fopen(
            "./cryptic.deciphered",
            "w");

    FeistelCipheringInitStream(
        &cipher,
        initVector);
    FeistelCipheringDecipherFile(
        &cipher,
        fpIn,
        fpOut);

    fclose(fpIn);
    fclose(fpOut);

    FeistelCipheringFreeStatic(&cipher);
    GSetFlush(&keys);
    printf("UnitTestFeistelStreamCipheringFile OK\n");
}

void UnitTestAll() {

    UnitTestFeistelCiphering();
    UnitTestFeistelStreamCipheringECB();
    UnitTestFeistelStreamCipheringCBC();
    UnitTestFeistelStreamCipheringCTR();
    UnitTestFeistelStreamCipheringFile();
    printf("UnitTestAll OK\n");

}

int main() {

    UnitTestAll();

    // Return success code
    return 0;

}

```

5 Unit tests output

```
Message:          072,101,108,108,111,032,087,111,114,108,100,046,  
Cyphered message: 118,073,094,092,063,132,192,196,201,204,246,068,  
Hello World.  
UnitTestFeistelCyphering OK  
UnitTestAll OK
```