Validación en CRC32 y encriptación en AES128

DESARROLLO DE SOFTWARE DE COMUNICACIONES EN AMBIENTES EMBEBIDOS (P2022_ESE007A)

Alberto Contreras Estrada
Email: a.contreras@iteso.mx

23/02/2022

PROGRAMA DE ESPECIALIDAD EN SISTEMAS EMBEBIDOS

www.sistemasembebidos.iteso.mx/alumnos

I T E S O A. C., Universidad Jesuita de Guadalajara

Periférico Sur Miguel Gómez Morín #8585, Tlaquepaque, Jalisco, México

Numero de Reporte Técnico: ESE-O2014-001

Objetivo: implementar una capa de protocolo que implemente la validación del mensaje recibido en CRC32 y encripte el mensaje con AES128.

Palabras Clave: CRC32, AES32, TCP/IP.

1. Tabla de contenido

2. Tabla de figuras

3. Introducción

CRC32 es una función para detectar errores que utiliza un algoritmo CRC32 que detecta cambios entre datos origen y destino. La función CRC32 convierte una cadena de longitud variable en una cadena de 8 caracteres que es una representación textual del valor hexadecimal de una secuencia binaria de 32 bits.

Advanced Encryption Standard (AES), es un esquema de cifrado por bloques adoptado como un estándar de cifrado por el gobierno de los Estados Unidos, creado en Bélgica. Estrictamente hablando, AES no es precisamente Rijndael (aunque en la práctica se los llama de manera indistinta) ya que Rijndael permite un mayor rango de tamaño de bloques y longitud de claves; AES tiene un tamaño de bloque fijo de 128 bits y tamaños de llave de 128, 192 o 256 bits, mientras que Rijndael puede ser especificado por una clave que sea múltiplo de 32 bits, con un mínimo de 128 bits y un máximo de 256 bits. La mayoría de los cálculos del algoritmo AES se hacen en un campo finito determinado. AES opera en una matriz de 4×4 bytes, llamada *state* (algunas versiones de Rijndael con un tamaño de bloque mayor tienen columnas adicionales en el state).

TCP/IP son las siglas de Transmission Control Protocol/Internet Protocol (Protocolo de control de transmisión/Protocolo de Internet). TCP/IP es un conjunto de reglas estandarizadas que permiten a los equipos comunicarse en una red como Internet.

Ethernet es la tecnología tradicional para conectar dispositivos en una red de área local (LAN) o una red de área amplia (WAN) por cable, lo que les permite comunicarse entre sí a través de un protocolo: un conjunto de reglas o lenguaje de red común. Ethernet describe cómo los dispositivos de red pueden formatear y transmitir datos para que otros dispositivos del mismo segmento de red de área local o de campus puedan reconocer, recibir y procesar la información. Un cable Ethernet es el cableado físico, encapsulado, por el que viajan los datos.

3.1. Requerimientos

- La nueva capa de protocolo debe proporcionar los siguientes servicios:
  - Validación de la integridad del mensaje con CRC32.
  - Cifrado del mensaje con AES128.
- Esta capa debe implementarse como una librería en lenguaje C y probarse comunicando la tarjeta K64 con un script de Python utilizando TCP/IP sobre Ethernet.
- La comunicación debe ser bidireccional, por lo que cada nodo debe de ser capaz de transmitir y recibir mensajes a través de la nueva capa.
- Para AES pueden utilizar la siguiente librería, o cualquier otra de su preferencia:
  - https://github.com/kokke/tiny-AES-c
- Para CRC pueden utilizar el ejemplo de CRC que viene con el SDK.
- Para probar la nueva capa de protocolo se debe implementar una aplicación que transmita al menos 8 paquetes diferentes (en tamaño y contenido) en ambas direcciones.

4. Descripción funcional

Se hace una conexión TCP mediante el puerto ethernet para una comunicación entre la PC y la K64f, elegí como cliente el script de Python (PC), y como servidor la K64F, se manda un mensaje desde la terminal de Python, este lo encripta, hace el CRC32, lo agrega paquete de datos al final y por último lo envía.
La K64f recibe este mensaje, le quita el checksum (CRC32), verifica que no haya perdida de datos, haciéndole el CRC32 y. si coincide con el checksum lo desencripta.
Ya que tiene el mensaje desencriptado, hace el mismo procedimiento que en el script de Python para mandar el mismo mensaje, a esto se le conoce como echo.

El primer reto de esta practica fue usar Python para usarlo como cliente o servidor, lo que aligero esto fue que los scripts de Python fueron proporcionados por el docente funcionando al 100%, modifique un poco los scripts para tener visualmente mas en claro como estaba haciendo el proceso de encriptación proporcionado en Python y estarlo imprimiendo al momento de ejecutarlo.

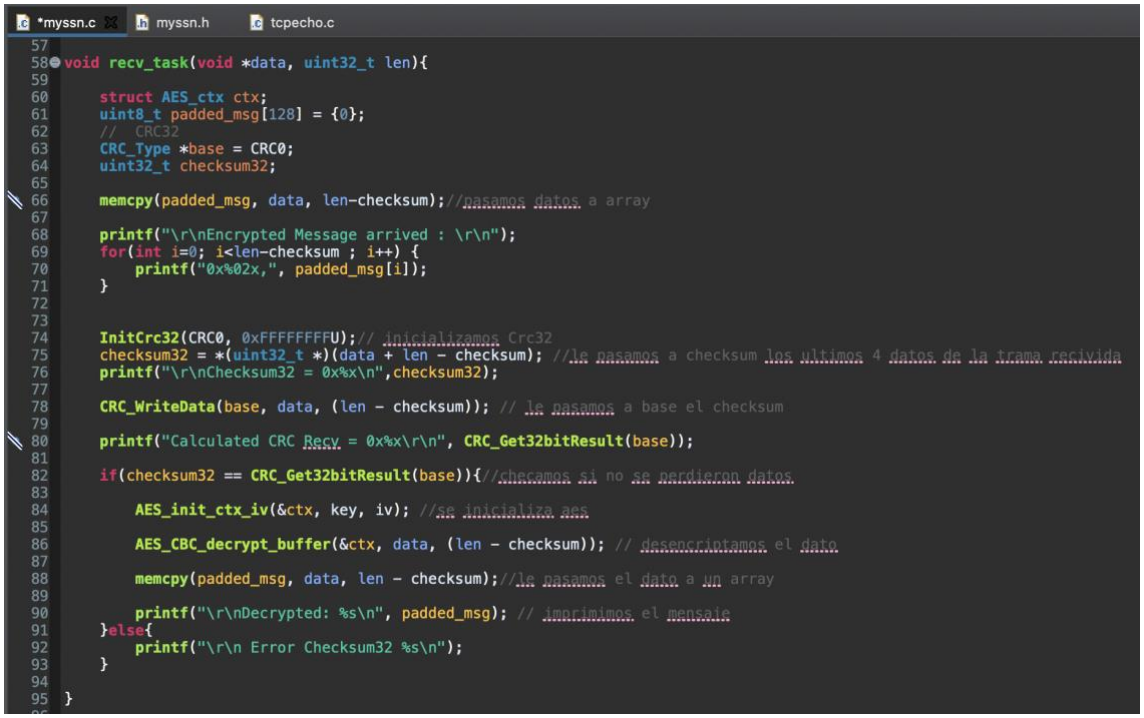
Para la implementación lo primero que hice fue crear myssn.c y .h en los cuales tiene definidas la función de recv_task y send_task, las cuales tiene como parámetro un apuntador al dato recibido, para esto tuve que agregar la inicialización del CRC32 que ya estaba proporcionada, KEY, IV. Imagen 3.1

myssn.c myssn.h tcpecho.c

```
1
2  #include "myssn.h"
3  #include <string.h>
4  #include <stdio.h>
5  #include "aes.h"
6  #include "fsl_crc.h"
7
8  #define checksum 4
9  uint8_t key[] = { 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06 };
10 uint8_t iv[]  = { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 };
11
12 static void InitCrc32(CRC_Type *base, uint32_t seed)
13 {
14     crc_config_t config;
15
16     config.polynomial         = 0x04C11DB7U;
17     config.seed               = seed;
18     config.reflectIn          = true;
19     config.reflectOut         = true;
20     config.complementChecksum = true;
21     config.crcBits            = kCrcBits32;
22     config.crcResult          = kCrcFinalChecksum;
23     CRC_Init(base, &config);
24 }
25
```

Figure 4.1 InitCrc32

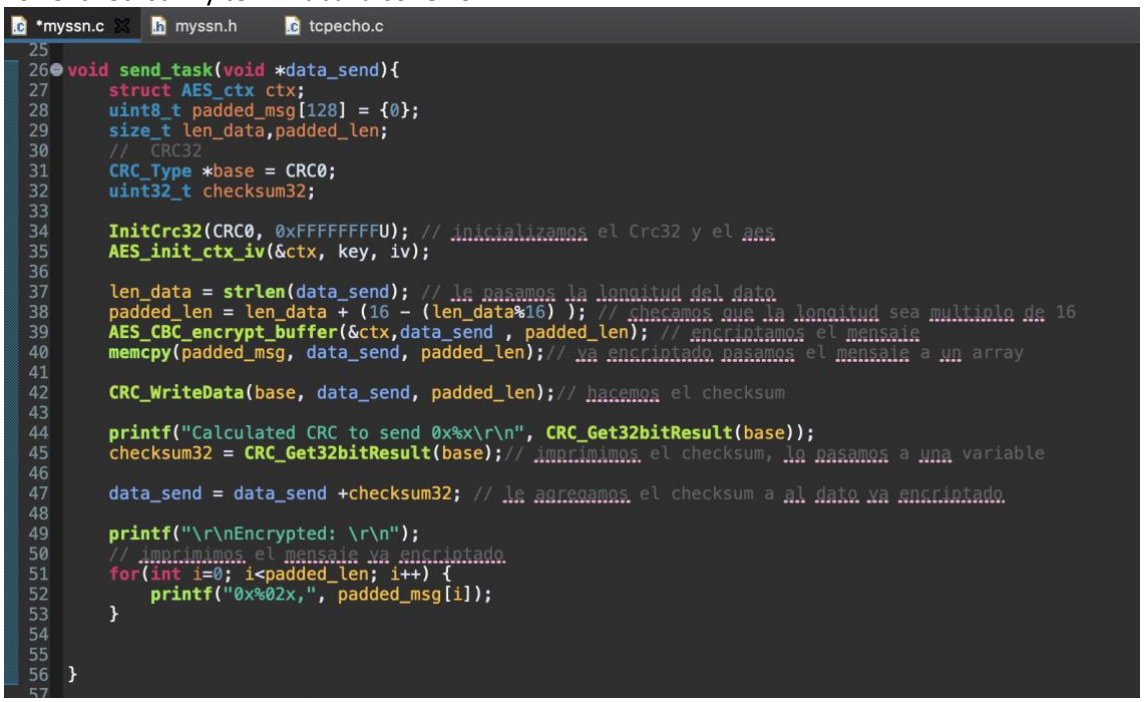
En la siguiente imagen primero le paso los datos recibidos a un array para la imprimir el mensaje recibido en hexadecimal, después inicializamos el CRC32, la línea de código 75 tuve que investigar mucho para saber como pasarle a un uint32_t el checksum del dato recibido, me encontré un ejemplo en internet y lo implementé y fue el resultado la línea 75. En la línea 78 le hacemos el RC32 para verificar que no haya perdida de datos, de lo contrario te manda un mensaje de error.
Si el mensaje no tiene errores proseguimos a inicializar aes, y después a desencriptar el mensaje para imprimirlo en la consola y cerciorarnos que llego correctamente.



```c
void recv_task(void *data, uint32_t len){

    struct AES_ctx ctx;
    uint8_t padded_msg[128] = {0};
    //  CRC32
    CRC_Type *base = CRC0;
    uint32_t checksum32;

    memcpy(padded_msg, data, len-checksum);//pasamos datos a array

    printf("\r\nEncrypted Message arrived : \r\n");
    for(int i=0; i<len-checksum ; i++) {
        printf("0x%02x,", padded_msg[i]);
    }


    InitCrc32(CRC0, 0xFFFFFFFFU);// inicializamos Crc32
    checksum32 = *(uint32_t *)(data + len - checksum); //le pasamos a checksum los ultimos 4 datos de la trama recivida
    printf("\r\nChecksum32 = 0x%x\n",checksum32);

    CRC_WriteData(base, data, (len - checksum)); // le pasamos a base el checksum

    printf("Calculated CRC Recv = 0x%x\r\n", CRC_Get32bitResult(base));

    if(checksum32 == CRC_Get32bitResult(base)){//checamos si no se perdieron datos

        AES_init_ctx_iv(&ctx, key, iv); //se inicializa aes

        AES_CBC_decrypt_buffer(&ctx, data, (len - checksum)); // desencriptamos el dato

        memcpy(padded_msg, data, len - checksum);//le pasamos el dato a un array

        printf("\r\nDecrypted: %s\n", padded_msg); // imprimimos el mensaje
    }else{
        printf("\r\n Error Checksum32 %s\n");
    }
}
```

Figure 4.2 Recv_task

En la función de send_task recibe como parámetro el dato desencriptado, volvemos a inicializar el aes ya que si no lo haces no funciona correctamente, después copie el código que nos proporciono el mentor tal cual para la encriptación, y al final al dato a mandar solo le agregue el checksum32 para completar la trama, por que si no mandaba el checksum Python recibía como “0” el checksum y terminaba la conexión.



```c
void send_task(void *data_send){
    struct AES_ctx ctx;
    uint8_t padded_msg[128] = {0};
    size_t len_data,padded_len;
    //  CRC32
    CRC_Type *base = CRC0;
    uint32_t checksum32;

    InitCrc32(CRC0, 0xFFFFFFFFU); // inicializamos el Crc32 y el aes
    AES_init_ctx_iv(&ctx, key, iv);

    len_data = strlen(data_send); // le pasamos la longitud del dato
    padded_len = len_data + (16 - (len_data%16) ); // checamos que la longitud sea multiplo de 16
    AES_CBC_encrypt_buffer(&ctx,data_send , padded_len); // encriptamos el mensaje
    memcpy(padded_msg, data_send, padded_len);// ya encriptado pasamos el mensaje a un array

    CRC_WriteData(base, data_send, padded_len);// hacemos el checksum

    printf("Calculated CRC to send 0x%x\r\n", CRC_Get32bitResult(base));
    checksum32 = CRC_Get32bitResult(base);// imprimimos el checksum, lo pasamos a una variable

    data_send = data_send +checksum32; // le agregamos el checksum a al dato ya encriptado

    printf("\r\nEncrypted: \r\n");
    // imprimimos el mensaje ya encriptado
    for(int i=0; i<padded_len; i++) {
        printf("0x%02x,", padded_msg[i]);
    }


}
```

Figure 4.3 Send_task

Por último en tcpecho.c la cual hace la conexión del socket en el apartado que esta esperando a que reciba algún dato (netconn_recv), ahí dentro mande llamar las funciones creadas para la des encriptación y encriptación de este dato para así reenviar el mismo dato para hacer el echo.

```
*myssn.c   myssn.h   tcpecho.c
#else /* LWIP_IPV6 */
  conn = netconn_new(NETCONN_TCP);
  netconn_bind(conn, IP_ADDR_ANY, 10000);
#endif /* LWIP_IPV6 */
  LWIP_ERROR("tcpecho: invalid conn", (conn != NULL), return;);

  /* Tell connection to go into listening mode. */
  netconn_listen(conn);

  while (1) {

    /* Grab new connection. */
    err = netconn_accept(conn, &newconn);
    /*printf("accepted new connection %p\n", newconn);*/
    /* Process the new connection. */
    if (err == ERR_OK) {
      struct netbuf *buf;
      void *data_recv;
      u16_t len_recv;

      while ((err = netconn_recv(newconn, &buf)) == ERR_OK) {

        do {

            netbuf_data(buf, &data_recv, &len_recv);// recive el bufer t la guarda en la variable data y guarda la longitd

            recv_task(data_recv,len_recv);//imprime el dato desencriptado y checa que no tenga perdidas de datos
            send_task(data_recv);// imprime el dato encriptao y listo para mandarlo

            err = netconn_write(newconn, data_recv, len_recv, NETCONN_COPY);// mandamos el mesaje encriptado

#if
            if (err != ERR_OK) {
              printf("tcpecho: netconn_write: error \"%s\"\n", lwip_strerr(err));
            }
#endif
        } while (netbuf_next(buf) >= 0);
        netbuf_delete(buf);
      }
      /*printf("Got EOF, looping\n");*/
      /* Close connection and discard connection identifier. */
      netconn_close(newconn);
      netconn_delete(newconn);
    }
  }
}
```

Trate de hacer la conexión en “myssn.c” pero al tener todo bien estructurado me mandaba a hardfault_Handler, le mostrare a continuación todas las modificaciones que hice, pero en la practica que enviare el que funciona sin esta modificación.

```
__attribute__ ((naked))
void HardFault_Handler(void){
    __asm(  ".syntax unified\n"
          // Check which stack is in use
            "MOVS   R0, #4  \n"
            "MOV    R1, LR  \n"
            "TST    R0, R1  \n"
            "BEQ    _MSP    \n"
            "MRS    R0, PSP \n"
            "B  _process      \n"
            "_MSP:  \n"
            "MRS    R0, MSP \n"
          // Load the instruction that triggered hard fault
          "_process:      \n"
            "LDR    R1,[R0,#24] \n"
            "LDRH   R2,[r1] \n"
          // Semihosting instruction is "BKPT 0xAB" (0xBEAB)
            "LDR    R3,=0xBEAB \n"
            "CMP    R2,R3 \n"
            "BEQ    _semihost_return \n"
          // Wasn't semihosting instruction so enter infinite loop
            "B  . \n"
```

Figure 4.4 HardFault_Handler

Cree la función socket_conect para hacer la conexión TCP, pero al momento de que regresaba como parámetro la estructura de la conexión se iba a ese hardfault.

```
struct netconn socket_conect(){
      struct netconn *conn, *newconn;
      err_t err;
      conn = netconn_new(NETCONN_TCP);
      netconn_bind(conn, IP_ADDR_ANY, 10000);
      netconn_listen(conn);

      while (1) {

        /* Grab new connection. */
        err = netconn_accept(conn, &newconn);
        /*printf("accepted new connection %p\n", newconn);*/
        /* Process the new connection. */
        if (err == ERR_OK) {
          break;
        }
      }
      return *newconn;
}
```

Figure 4.5 Socket_Conect

También me di cuenta al termino de la función recv_task tendría que llamar a la función send_task, ya que en la función cree el WHILE para estar recibiendo los datos que entraban, en la figura 3.6 se observa la modificación que hice.

```
void recv_task(struct netconn *conn){
    struct AES_ctx ctx;
    uint8_t padded_msg[128] = {0};
    struct netbuf *buf;
    err_t err;
    void *data;
    uint16_t len;
    //  CRC32
    CRC_Type *base = CRC0;
    uint32_t checksum32;

     while ((err = netconn_recv(conn, &buf)) == ERR_OK) {

            do {
                netbuf_data(buf, &data, &len);

                memcpy(padded_msg, data, len-checksum);//pasamos datos a array

                printf("\r\nEncrypted Message arrived : \r\n");
                for(int i=0; i<len-checksum ; i++) {
                    printf("0x%02x,", padded_msg[i]);
                }


                InitCrc32(CRC0, 0xFFFFFFFFU);// inicializamos Crc32
                checksum32 = *(uint32_t *)(data + len - checksum); //le pasamos a checksum los ultimos 4 datos
                printf("\r\nChecksum32 = 0x%x\n",checksum32);

                CRC_WriteData(base, data, (len - checksum)); // le pasamos a base el checksum

                printf("Calculated CRC Recv = 0x%x\r\n", CRC_Get32bitResult(base));

                if(checksum32 == CRC_Get32bitResult(base)){//checamos si no se perdieron datos

                    AES_init_ctx_iv(&ctx, key, iv); //se inicializa aes

                    AES_CBC_decrypt_buffer(&ctx, data, (len - checksum)); // desencriptamos el dato

                    memcpy(padded_msg, data, len - checksum);//le pasamos el dato a un array

                    printf("\r\nDecrypted: %s\n", padded_msg); // imprimimos el mensaje
                    send_task(conn,data);

                }else{
                    printf("\r\n Error Checksum32 %s\n");
                }
            } while (netbuf_next(buf) >= 0);
              netbuf_delete(buf);
            }
}
```

Figure 4.6 Recv_task_v2

La modificación que hice en el “tcpecho.c” fue la siguiente:

```
static void
tcpecho_thread(void *arg)
{
  struct netconn *conn;
  err_t err;
  LWIP_UNUSED_ARG(arg);



    *conn=socket_conect();
    recv_task(conn);//imprime el dato desencriptado y checa que no tenga perdidas de datos


}
```

Figure 4.7 Tcpecho_threat

En la función de send_task solo agregue la función que envía el dato por la conexión.

```
void send_task(struct netconn *conn,void *data_send){
    struct AES_ctx ctx;
    uint8_t padded_msg[128] = {0};
    size_t len_data,padded_len;
    //  CRC32
    CRC_Type *base = CRC0;
    uint32_t checksum32;

    InitCrc32(CRC0, 0xFFFFFFFFU); // inicializamos el Crc32 y el aes
    AES_init_ctx_iv(&ctx, key, iv);

    len_data = strlen(data_send); // le pasamos la longitud del dato
    padded_len = len_data + (16 - (len_data%16) ); // checamos que la longitud sea multiplo de 16
    AES_CBC_encrypt_buffer(&ctx,data_send , padded_len); // encriptamos el mensaje
    memcpy(padded_msg, data_send, padded_len);// ya encriptado pasamos el mensaje a un array

    CRC_WriteData(base, data_send, padded_len);// hacemos el checksum

    printf("Calculated CRC to send 0x%x\r\n", CRC_Get32bitResult(base));
    checksum32 = CRC_Get32bitResult(base);// imprimimos el checksum, lo pasamos a una variable

    data_send = data_send +checksum32; // le agregamos el checksum a al dato ya encriptado

    printf("\r\nEncrypted: \r\n");
    // imprimimos el mensaje ya encriptado
    for(int i=0; i<padded_len; i++) {
        printf("0x%02x,", padded_msg[i]);
    }

    netconn_write(conn, data_send, padded_len, NETCONN_COPY);// mandamos el mesaje encriptado

}
```

Figure 4.8 Send_task_v2

5. Resultados

Para los siguientes resultados mande varios mensajes los cuales se observan en las siguientes capturas en la terminal del MCUExpresso y la terminal de Python.

Mensajes enviados desde el cliente: 1ro mensaje – 10, 2do mensaje – 120, 3er mensaje-360, 4to mensaje – 480.

Figure 5.1 Python_1

Figure 5.2 Python_2

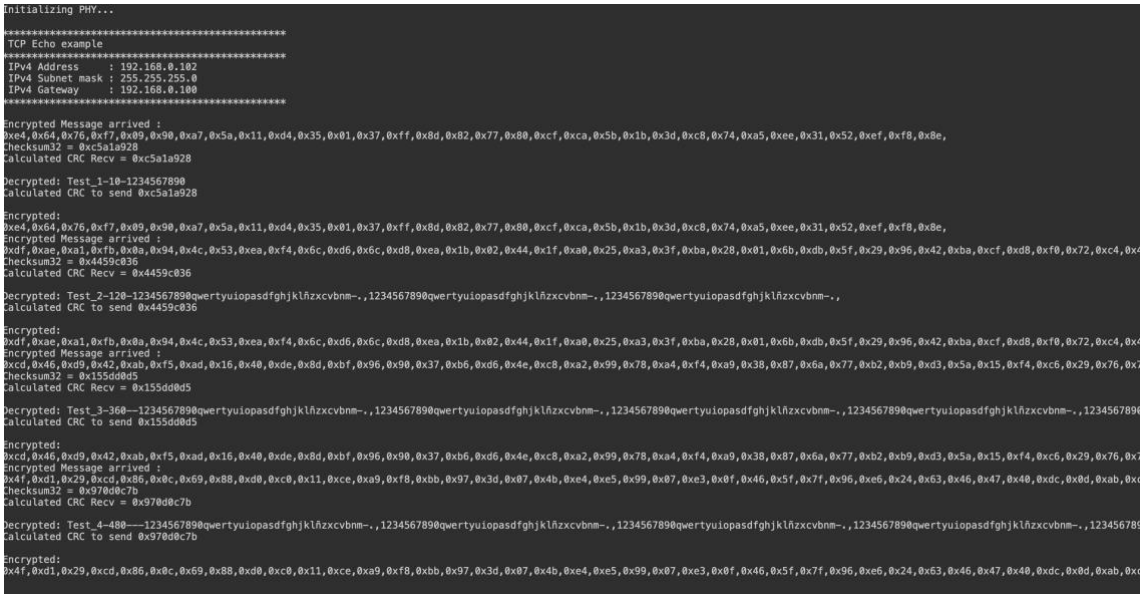


Figure 5.3 MCUExpresso

6. Conclusión

Para la implementación la fui dividiendo y fui imprimiendo lo que me iba resultando, y así fue mas fácil detectar los errores, en cambio anteriores veces planteaba mi idea y tenia que ir mirando después parte por parte donde estaba el error, el reto para mi fue al momento de recibir el mensaje y guardar el checksum en una variable ya que el dato que recibía era una trama larga y tenia que castear a 32bits los últimos 8bytes de la trama recibida. Algo no tenia muy clara y con esta practica reforcé el conocimiento fue con el callback, me di cuenta de que es muy sencillo de usar y muy útil. Nunca había usado Python, tuve complicaciones al instalarlo en mi ordenador y al comprender el código, pero termino siendo algo sencillo ya que te enfocás en ello.
Se me complico la implementación de la capa del protocolo ya que no me quedaba claro lo que se tenia que hacer para llegar a ese objetivo, al final me di cuenta de lo que se tenia que lograr, pero no tuve el tiempo suficiente para lograrlo, me que quede en un hardfault que no supe por que me lo daba, lo importante fue que me quedo claro el objetivo de esta practica pero no logre que funcionara correctamente con esa implementación.