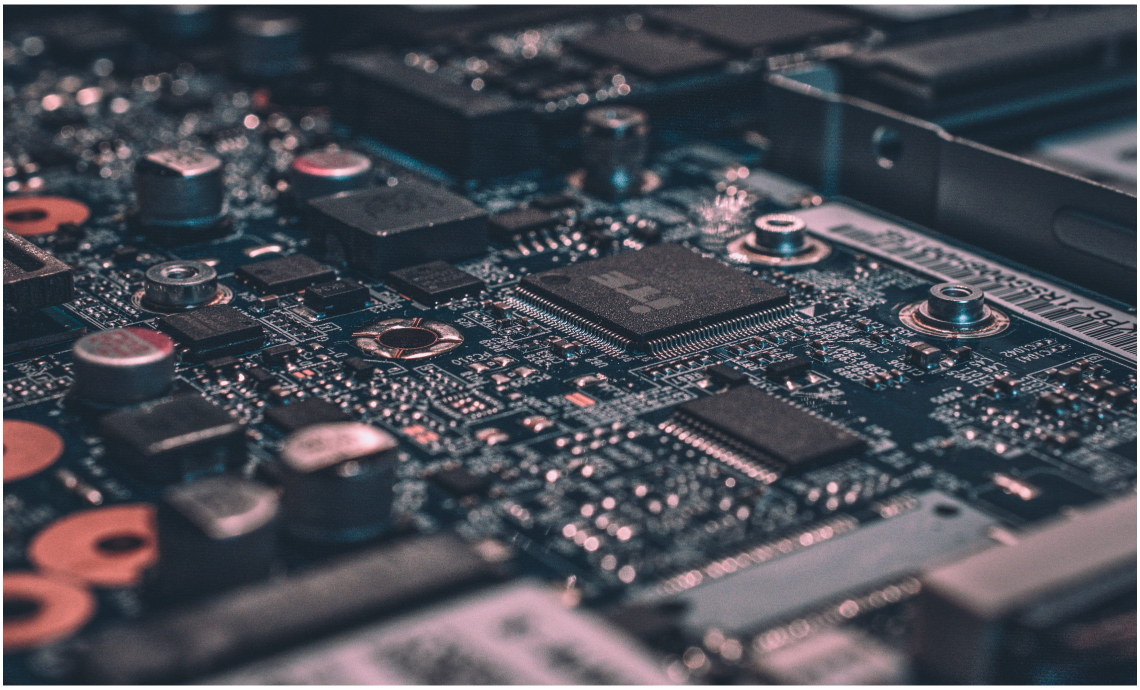
DIC - Project 2

Zephyr RTOS Crypto Processor

by Dorde Stojanovic



hand out on: .... April 17, 2021
supervisor: .... Lezuo Roland, DI
student: .... Stojanovic Dorde
year: 2020/21
class: 5BHEL

HTBLVA Anichstraße

Contents

1 General

1.1 Abstract

Simply put, the task is to implement a cryptoprocessor in Zephyr RTOS. The processor should successfully complete the tests given in the specification. The virtual serial interface is used to address the cryptoprocessor. This is decrypted using AES-128 (Advanced-Encryption-Standard-128-bit). All this is implemented on the operating-system-version 2.4.0. In this project, a 64-bit Linux microcontroller board ("native_posix_64") is used. The native_posix_64 board implements a virtual serial interface called "UART_0". At the same time, a Crypo API called "CRYPTO_TC" is implemented using libtinycrypt. Both implemented drivers are to be used.

2 Prior Knowledge and Component Information

2.1 Specifications

MAIN_Thread

The main thread of the programm should start all other threads and then only sporadically give a life message.

UART_IN_Thread

The native_posix_64 UART driver does not support IRQ, so reading is blocking. Therefore a thread is needed that reads and as soon as it has a character it sends it to a "Message_Queue". The processing threads consume this character from there. When the character "." has been read, it should be echoed back to the serial port. If the processing thread is busy, the message "BUSY" is output via the serial interface.

UART_OUT_Thread

This thread ensures that multiple threads can output messages to the serial port without interfering with each other. To do this, it reads the messages to be sent from a queue and sends it character by character via the serial interface before it sends another before fetching another message from its queue.

PROCESSING_Thread

This thread performs the crypto operations. These can take a very long time. The processing thread can store exactly one message in its message queue, This is because one is in processing, one waiting before the system has to generate "BUSY" messages.

2.2 Serial Protocol

Serial connection possible (= alive)

Whenever a '.' is received, a '.' is immediately returned.

Cryptoprocessor available (= avail)

When a 'P' is received, the processing thread responds with "PROCESSING AVAIL".

Load key (= key)

A 'K' followed by 16 bytes of AES-128 key and any character (discarded) loads a new key into the cryptoprocessor.

Load Initial Vecotr (= iv)

An 'I' followed by 16 bytes of AES-128 key and an arbitrary character (discarded) loads a new key into the cryptoprocessor

Decrypt in CBC Mode

A 'D' followed by a byte length (ciphertext, must be multiple of cipherblocksize) followed by correspondingly long ciphertext and any character (discarded). The cryptoprocessor decrypts with the key/iv in aes128-cbc and returns the plaintext on the serial interface. "XERROR" if errors have occurred (e.g.: length).

Further Information

All further information regarding the start of the test suite and the start of the crypto processor can be found in the Markdown file (readme.md) provided by the supervisor.

2.3 Zephyr (real time operating system)

2.3.1 General infromation

"The Zephyr OS is based on a small-footprint kernel designed for use on resource-constrained and embedded systems: from simple embedded environmental sensors and LED wearables to sophisticated embedded controllers, smart watches, and IoT wireless applications." [1]

2.3.2 Features

"Zephyr intends to provide all components needed to develop resource-constrained and embedded or microcontroller-based applications. This includes, but is not limited to:

- A small kernel
- A flexible configuration and build system for compile-time definition of required resources and modules
- A set of protocol stacks (IPv4 and IPv6, OMA LWM2M, MQTT, 802.15.4, Bluetooth Low Energy, CAN)
- A virtual file system interface with several flash file systems for non-volatile storage
- Management and device firmware update mechanisms" [2]

2.3.3 West

"The Zephyr project includes a swiss-army knife command line tool named west1. West is developed in its own repository. West's built-in commands provide a multiple repository management system with features inspired by Google's Repo tool and Git submodules. West is also "pluggable": you can write your own west extension commands which add additional features to west. Zephyr uses this to provide conveniences for building applications, flashing and debugging them, and more." [3]

2.3.4 Ninja

Ninja is a tool designed for speed optimisation and contains only the most necessary functions to describe arbitrary dependency graphs. Due to the lack of syntax, it is not possible to express complex decisions. Instead, Ninja is intended to be used with a separate program that generates the input files. Autotools mean that a lot of factors have to be taken into account for the build time. Ninja build files allow Ninja to evaluate incremental builds quickly.

[1] Source: 17.04.2021, https://docs.zephyrproject.org/latest/introduction/index.html
[2] Source: 17.04.2021, https://en.wikipedia.org/wiki/Zephyr_(operating_system)
[3] Source: 17.04.2021, https://docs.zephyrproject.org/latest/guides/west/index.html

2.3.5 Kconfig

"KConfig is a selection-based configuration system originally developed for the Linux kernel. It is commonly used to select build-time options and enable or disable features."[4]

Through a curses-based or graphical menu interface, most users interact with KConfig. This is usually invoked by running "menuconfig" or "guiconfig". In this interface, the user selects the desired options and functions and saves a configuration file, which is then used as input for the build process. The Zephyr kernel and subsystems can be configured at build time to suit specific application and platform requirements. Configuration is done through Kconfig, the same configuration system used by the Linux kernel. The main goal is to provide configuration support without changing the source code.

2.3.6 Devicetree

The Zephyr RTOS has a devicetree, a data structure that simplifies the hardware description. As soon as a project is compiled here, a ".dts" file is automatically and immediately created which corresponds to the devicetree. However, this also speeds up the process, as the data structure of all components is simply transferred during booting and thus everything does not have to be described in detail.

2.4 Threads

Threads enable parallelism in programmes that would otherwise be executed sequentially. The thread itself is executed sequentially. Threads are, so to speak, processes that are executed line by line. However, it is possible to connect threads in parallel and ensure that the entire programme is no longer executed sequentially but in parallel. A symbolic structure of a programme with threads looks like this:

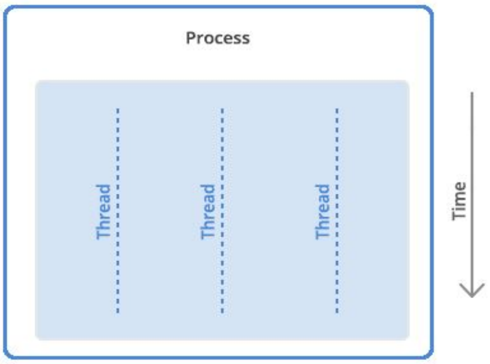


Figure 1: Process Structure
(17.04.2021,
https://www.backblaze.com/blog/whats-the-diff-programs-processes-and-threads/)

[4]Source: 17.04.2021, https://docs.legato.io/19_11/toolsKconfig.html

2.5 Message Queues

"A message queue is a kernel object that implements a simple message queue, allowing threads and ISRs to asynchronously send and receive fixed-size data items. Any number of message queues can be defined (limited only by available RAM). Each message queue is referenced by its memory address. A message queue has the following key properties:

- A ring buffer of data items that have been sent but not yet received.
- A data item size, measured in bytes.
- A maximum quantity of data items that can be queued in the ring buffer."[5]

[5]Source: 17.04.2021, https://docs.zephyrproject.org/latest/reference/kernel/data_passing/message_queues.html

3 Implementation

3.1 Github - Repo

https://github.com/Djordje-Stojanovic/FSST/tree/main/Stojanovic_DIC_PROJECT_2

3.2 Code

Disclaimer: The basis on which I built the code was Patrick Wintners implementation idea. He helped me with the structure.

3.2.1 Main

```
//-----------------------------------------------------------------------------
//this is the main function
void main(void)
{
        // Initializations
        uart_dev = device_get_binding(DT_LABEL(UART_DEVICE));

    // If uart is not found
        if(!uart_dev)
    {
        //print that it couldnt be found
                printk("UART could not be found\n");
                return;
        }
    //else print that UART was found
        printk("UART has been found\n");

        //Configuration of the UART
        uartconf.baudrate = 9600; //initialization of a fixed baudrate
        uartconf.parity = UART_CFG_PARITY_NONE; //initialization of a fixed parity
        uartconf.stop_bits = UART_CFG_STOP_BITS_1; //initialization of stop bits
        uartconf.data_bits = UART_CFG_DATA_BITS_8; //initialization of data bits
        uartconf.flow_ctrl = UART_CFG_FLOW_CTRL_NONE; //initialization of flow control (none)

    //If UART couldnt be configured
        if(!uart_configure(uart_dev, &uartconf))
    {
        //Print that it could not be configured
                printk("Configuration of UART failed\n");
                return;
        }
    //else print that it could been configured
        printk("UART configured\n");

        for(;;)
    {
                printk("\nmain is waiting for death\n");
                k_msleep(10*1000); // 10s time sleep
        }
}
//-----------------------------------------------------------------------------
```

Listing 1: Stojanovic_DIC_Project2.c - Main

3.2.2 Initializations

```c
//------------------------------------------------------------------------------
//Definition of constant variables:
//For the UART:
#define UART_DEVICE DT_NODELABEL(uart0) //Nodelable according to Zephyr Documentation is used for
                                        // "Get a node identifier for a node label."
#define sizeOfStack 1024 //Reporesents the Stack Size for the Thread
#define preference 0 //Represents the preference of the thread
//------------------------------------------------------------------------------


//------------------------------------------------------------------------------
//Further Initialization:
enum {init, avail}; //initialize enum
int state = init; //initialize state
const struct device *uart_dev; //initialize the struct "device"
struct uart_config uartconf; ////initialize the struct "uart_config"

//Initialization for Uart
void uartInput(void *, void *, void *); //Input of the UART
void uartOutput(void*, void*, void*); //Output of the UART
void uartProcess(void*, void*, void*); //Process of the UART
//------------------------------------------------------------------------------


//------------------------------------------------------------------------------
//Thread Initialization:
K_THREAD_DEFINE(uartInputThreadIdentifier, sizeOfStack, //Input Thread initialization
                uartInput, NULL, NULL, NULL,
                preference, 0, 0);
K_THREAD_DEFINE(uartOutputThreadIdentifier, sizeOfStack, //Output Thread initialization
                uartOutput, NULL, NULL, NULL,
                preference, 0, 0);
K_THREAD_DEFINE(uartProcessThreadIdentifier, sizeOfStack, //Process Thread initialization
                uartProcess, NULL, NULL, NULL,
                preference, 0, 0);
//------------------------------------------------------------------------------


//------------------------------------------------------------------------------
// Message Queue Initialization:
K_MSGQ_DEFINE(uartMessageQueue, 100*sizeof(char), 10, 1); //message Queue inizialization
K_MSGQ_DEFINE(uartProcessMessageQueue, 100*sizeof(char), 10, 1); //message Queue Process initialization
//------------------------------------------------------------------------------
```

Listing 2: Stojanovic_DIC_Project2.c - Initialization

3.2.3 Input-Function

```c
//------------------------------------------------------------------------------
//this function is responsible for the uart input
//there are 3 possible cases which will be stated bellow
void uartInput(void *firstPointer, void *secondPointer, void *thirdPointer)
{

        // Initializations
        ARG_UNUSED(firstPointer);
        ARG_UNUSED(secondPointer);
        ARG_UNUSED(thirdPointer);
        unsigned char input;

    //next thing is an infinite loop
        for(;;)
    {
                switch(state)
        {
            //if state is init:
                        case init:
                                if(!uart_poll_in(uart_dev, &input))
                { //if data is recieved, the following is printed.
                                        printk("Data has been recieved: %c\n", input);
                                        switch(input)
                    {
                        //if "." is recieved we should print "." back
                                                case '.':
                                                        k_msgq_put(&uartMessageQueue, ".\n", K_FOREVER);
                                                        break;
                        //else if we got "P" then we should change the state to avail
                                                case 'P':
                                                        printk("Sate is beeing changed to avail\n");
                                                        state = avail;
                                                        break;
                        //if we dont get P or . then it should break
                                                default: break;
                                        }
                                }
            //if state is avail:
                        case avail: break; //the function is breaked
                        default: break;
                }
                k_msleep(1); //According to Zephyr documentation:
                     // "This routine puts the current thread to sleep for -duration- milliseconds."
        }
        return;
//------------------------------------------------------------------------------
}
```

Listing 3: Stojanovic_DIC_Project2.c - Input-Function

3.2.4 Output-Function

```c
//---------------------------------------------------------------------------
//this function is responsible for the uart output
void uartOutput(void *firstPointer, void *secondPointer, void *thirdPointer)
{

        // Initializations
        ARG_UNUSED(firstPointer);
        ARG_UNUSED(secondPointer);
        ARG_UNUSED(thirdPointer);
        unsigned char *output=malloc(100*sizeof(char));

    //next thing is an infinite loop
        for(;;)
    {
        //resets the memmory
                memset(output, 0, strlen(output));
        //Scan the message queue, if it is 0 then
                if(k_msgq_get(&uartMessageQueue, output, K_NO_WAIT)==0)
        {
            //send data
                        printk("Sending the data: <%s>\n", output);

                        for(int i=0; i<strlen(output); i++)
            {
                                uart_poll_out(uart_dev, *(output+i));
                //prints out the sent data
                                printk("Data that has been sent: <%x>\n", *(output+i));
                        }
                }
                k_msleep(1); //According to Zephyr documentation:
                     // "This routine puts the current thread to sleep for -duration- milliseconds."
        }
        return;
}
//---------------------------------------------------------------------------
```

Listing 4: Stojanovic_DIC_Project2.c - Output-Function

3.2.5 Process-Function

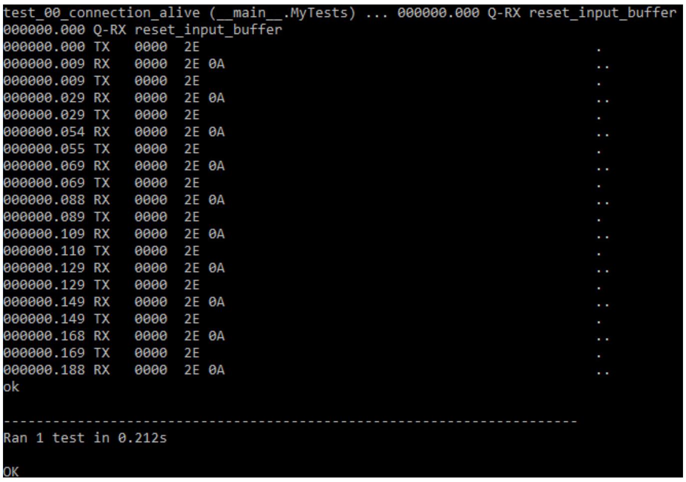
```c
//---------------------------------------------------------------------------
//this function is responsible for the uart process in general
void uartProcess(void *firstPointer, void *secondPointer, void *thirdPointer)
{
        // Initializations
        ARG_UNUSED(firstPointer);
        ARG_UNUSED(secondPointer);
        ARG_UNUSED(thirdPointer);

    //next thing is an infinite loop
        for(;;)
    {
                switch(state)
        {
            //here the process should break if it is in state init
                        case init:
                                break;
            //if it is in state avail it should make sure to tell that processing is now avaliable
                        case avail:
                                k_msgq_put(&uartMessageQueue, "PROCESSING AVAILABLE\n", K_FOREVER);
                //telling that its gonna change state
                                printk("Changing state the state back to init\n");
                //changing state
                                state = init;
                                break;
                        default: break;
                }
                k_msleep(1); //According to Zephyr documentation:
                     // "This routine puts the current thread to sleep for -duration- milliseconds."
        }
        return;
}
//---------------------------------------------------------------------------
```

Listing 5: Stojanovic_DIC_Project2.c - Process-Function

4 Test of Function

4.1 Test_00_connection_alive

This test has been passed sucessfuly:

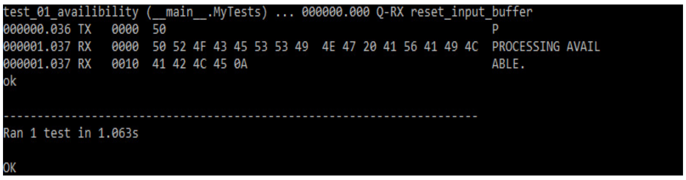


```
test_00_connection_alive (__main__.MyTests) ... 000000.000 Q-RX reset_input_buffer
000000.000 Q-RX reset_input_buffer
000000.000 TX   0000  2E                                                       .
000000.009 RX   0000  2E 0A                                                    ..
000000.009 TX   0000  2E                                                       .
000000.029 RX   0000  2E 0A                                                    ..
000000.029 TX   0000  2E                                                       .
000000.054 RX   0000  2E 0A                                                    ..
000000.055 TX   0000  2E                                                       .
000000.069 RX   0000  2E 0A                                                    ..
000000.069 TX   0000  2E                                                       .
000000.088 RX   0000  2E 0A                                                    ..
000000.089 TX   0000  2E                                                       .
000000.109 RX   0000  2E 0A                                                    ..
000000.110 TX   0000  2E                                                       .
000000.129 RX   0000  2E 0A                                                    ..
000000.129 TX   0000  2E                                                       .
000000.149 RX   0000  2E 0A                                                    ..
000000.149 TX   0000  2E                                                       .
000000.168 RX   0000  2E 0A                                                    ..
000000.169 TX   0000  2E                                                       .
000000.188 RX   0000  2E 0A                                                    ..
ok

----------------------------------------------------------------------
Ran 1 test in 0.212s

OK
```

Figure 2: Test: 00_connection_alive

4.2 Test_01_availibility

This test has been passed sucessfuly:



```
test_01_availibility (__main__.MyTests) ... 000000.000 Q-RX reset_input_buffer
000000.036 TX   0000  50                                                P
000001.037 RX   0000  50 52 4F 43 45 53 53 49  4E 47 20 41 56 41 49 4C  PROCESSING AVAIL
000001.037 RX   0010  41 42 4C 45 0A                                    ABLE.
ok

----------------------------------------------------------------------
Ran 1 test in 1.063s

OK
```

Figure 3: Test: 01_availibility