

# Serial Crypto Processor Using Zephyr

David Rieser  
Supervisor : Prof. Lezuo

April 12, 2021

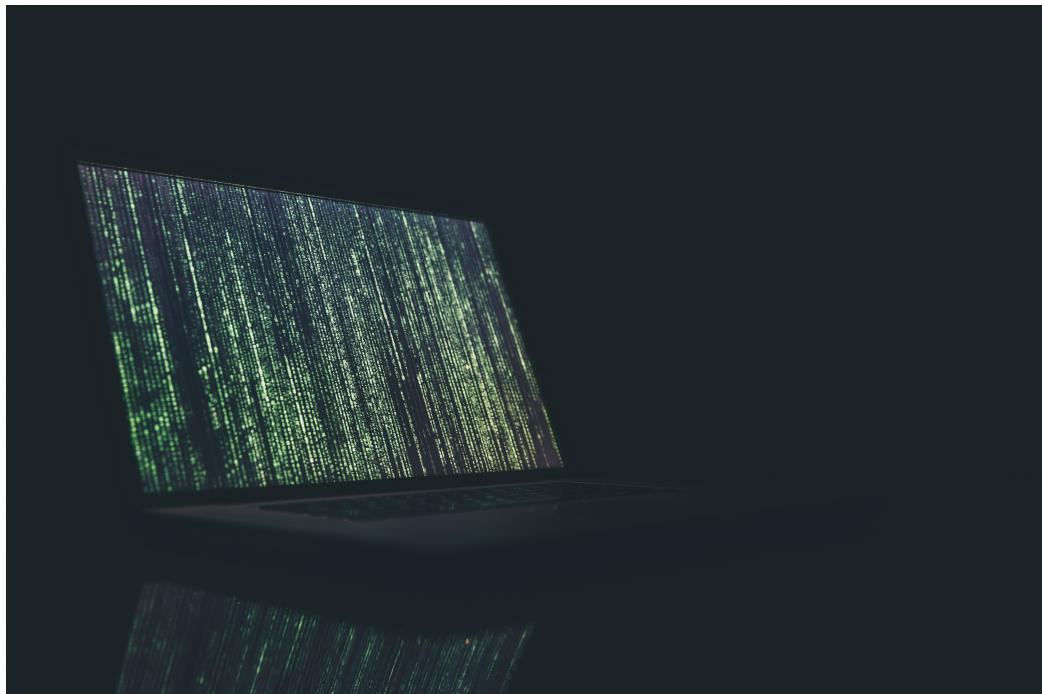


Figure 1: Photo by Markus Spiske on Unsplash

# Contents

<b>1</b>	<b>Project Requirements</b>	<b>1</b>
<b>2</b>	<b>Project Advantages and Disadvantages</b>	<b>2</b>
2.1	Zephyr . . . . .	2
2.2	West . . . . .	2
2.3	KConfig . . . . .	3
2.4	CMake . . . . .	3
2.5	Ninja . . . . .	3
2.5.1	Device Tree . . . . .	3
<b>3</b>	<b>Program Implementation</b>	<b>4</b>
3.0.1	Threads . . . . .	4
3.0.2	Message-Queues . . . . .	4
3.1	Linux Pseudo-Terminals . . . . .	4
3.2	TinyCrypt . . . . .	5
<b>4</b>	<b>Project Execution</b>	<b>6</b>
4.1	Block-Diagram . . . . .	6
4.2	Build-Settings . . . . .	7
4.3	Initialisation . . . . .	8
4.3.1	Message-Queue Initialisation . . . . .	8
4.3.2	UART_0 Initialisation . . . . .	8
4.3.3	Crypto-Device Initialisation . . . . .	10
4.3.4	Thread Initialisation . . . . .	11
4.4	UART-In-Thread . . . . .	11
4.5	UART-Out-Thread . . . . .	13
4.6	Processing Thread . . . . .	14
4.7	Encryption and Decryption . . . . .	14
4.7.1	Input and Output Buffers . . . . .	14
4.7.2	Implementation . . . . .	15
4.8	Unit-Tests . . . . .	16

## Figures

Statemachine . . . . .	1
Message-Queue Example Diagram . . . . .	4
Serial Crypto Processor Block Diagram . . . . .	6

## Listings

1 CMakelists.txt . . . . .	7
2 prj.conf . . . . .	7
3 Makefile.posix . . . . .	7
4 Message Queue Initialisation . . . . .	8
5 UART 0 Initialisation . . . . .	8
6 UART-0 Configuration . . . . .	9
7 Crypto-Device Initialisation . . . . .	10
8 Crypto Hardware Capability Check . . . . .	10
9 Thread Initialisation . . . . .	11
10 State Definitions . . . . .	11
11 State Machine Pseudo-Code . . . . .	12
12 Message Struct Definition . . . . .	13
13 UART Out Thread Pseudo-Code . . . . .	13
14 Processing Thread Pseudo-Code . . . . .	14
15 Global Buffers for Encryption and Decryption . . . . .	14
16 Encryption and Decryption Pseudo-Code . . . . .	15

# 1 Project Requirements

The Project consists of the creation of a Processor which can receive data, then encrypt or decrypt this data using AES-128 and then send it back via the Serial-Interface.

The Processor should be developed using the [native\\_posix-Board](#). Using the [native\\_posix-Board](#) the Program can be compiled into a normal executable which can be run on the Host-System (e.g. Linux).

The Serial-Interface should be implemented using the UART-Inteface which will connect to /dev/pts/0 when the Board is started.

The Crypto-Processor should run using a predefined Statemachine and is to be tested using a Python Unit-Test-Script.

The Cryptographic Operations (AES-128 Encryption and Decryption) should be implemented using the Tinycrypt-Crypto-Device.

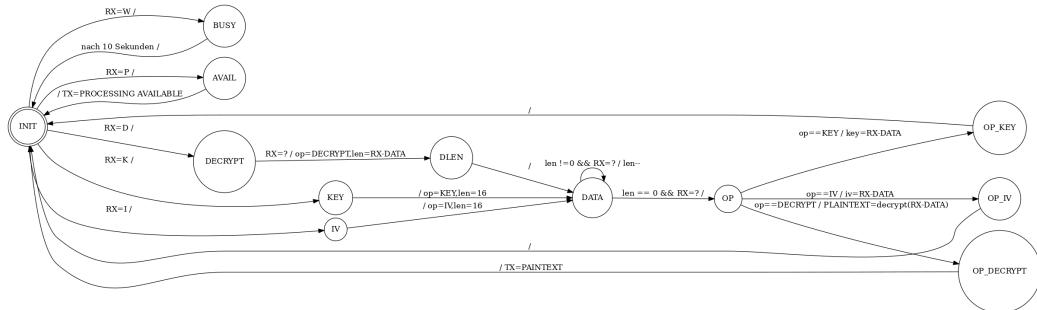


Figure 2: Statemachine

## 2 Project Advantages and Disadvantages

### 2.1 Zephyr

Zephyr is a RTOS (Real-Time-Operating-System).

An OS (Operating System) handles the Low-Level Functions and Execution, enabling Developers to write Code which can be run independent of the actual Hardware. OS's are in charge of Memory-, Process- and Device-Management.

A RTOS in Contrast to a normal OS removes the randomness of Task Scheduling making the System more reliable, by placing Constraints on when each Process is executed. In a normal OS most Processes switch randomly (Round-Robin) while in RTOS every Process can be interrupted by another Process with a higher Priority (Preemptive Scheduling).

[Introduction to RTOS](#)

Zephyr is designed for Resource-Constrained Devices and is build using the “West”-Meta-Tool.

It is developed by [The Linux Foundation](#) and released under the [Apache License 2.0](#).

Zephyr 2.4.0 was used for this Project. It was installed for WSL2 on Windows 10 using the [official Getting-Started-Guide from Zephyr](#)

### 2.2 West

West is Zephyr’s Meta-Tool.

It oversees the Build-Process and manages the Repository.

West makes sure the right Versions of the Zephyr-Libraries are installed and passed to the Build-System.

It replaces Package-Managers like [apt](#) or [Conan](#).

West also works in Conjunction with KConfig, CMake and Ninja to build the actual Application using the Zephyr SDK’s (Software Development Kit).

## 2.3 KConfig

[KConfig](#) is a configuration system originally developed for the Linux-Kernel which can be used to enable or disable features or to select build-time-Options.

West looks at the KConfig-File before Building and moves the specified Dependencies to the Build-Directory so the Build-System can use them.

## 2.4 CMake

[CMake](#) is a Software Development Tool which can generate Build Files for other Build-Systems or other Build-Tools.

West uses CMake to generate Build-Files for Ninja.

CMake can also be used to find Dependencies for Source-Code, testing Executables, and packaging Releases.

## 2.5 Ninja

[Ninja](#) is a Build-Tools which is designed to be as fast as possible and to have it's Build-Files created by a higher-level Build-System (like CMake).

Ninja is meant to replace the conventional “Make“-Build-System which is slow when building big Projects with lots of Files.

### 2.5.1 Device Tree

The [Device Tree](#) is a way of defining hardware and configuration Information for Zephyr-Boards.

It is used so Code can be dynamically included or excluded for every Board without hard-coding every device into the OS.

It removes the need for hardcoded Addresses inside the Libraries and enables that Libraries can be reused across a range of different devices with different Hardware.

### 3 Program Implementation

The Project's Goal is to create a Encryption and Decrytion device which doesn't need a lot of resources and can be easily accessed using UART.

The Program is implemented using a Statemachine and seperate Message-Queues per Thread which enables it to be modified very quickly.

The Threads each have a dedicated task and complement each other, which creates an efficient system as a whole, but require some baseline hardware to work.

The implementation on the [native\\_posix-Board](#) has one big drawback which is that Interrupts cannot be implemented.

This means the Program had to be developed using non-blocking Calls.

If the Program is to be implemented on a proper device these can be rewritten easily using the equivalent blocking statements.

#### 3.0.1 Threads

A Thread is a sequence of instruction which can be run independent of it's parent-process (the main-Thread in this case).

This allows the CPU to work on seperate Jobs simultaneously.

#### 3.0.2 Message-Queues

Message Queues are FIFO-Buffers (First-In-First-Out-Buffers).

Buffers store data for later use and First-In-First-Out means that the data which came in first comes out first.

FIFO-Buffers can deliver data between Threads in a safe manner.

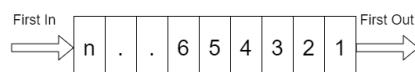


Figure 3: Message-Queue Example Diagram

#### 3.1 Linux Pseudo-Terminals

PTY's also called [Pseudo-Terminals](#) are bidirectional communication channels. They enable Programs to transfer Data between one another and appear as a normal File under `/dev/pts/`.

### 3.2 Tinycrypt

Tinycrypt is a small footprint cryptography library written in Assembler and implemented in Zephyr.

It is developed and maintained by [Intel's Open Souce Technlogy Center](#).

Tinycrypt enables devices with limited resources to encrypt and decrypt data with a variety of Algorithms at a good pace because it is implemented using Assembler.

Tinycrypt's downside is that it needs it's memory setup in a particular manner where the buffers are contingous.

Tinycrypt also only allows one Crypto-Operation at the same Time per device which limits the scalability if no additional Hardware is provided.

## 4 Project Execution

The Complete Codebase for the Project can be found on: [Github](#)

### 4.1 Block-Diagram

The Serial-Crypto Processor is split into four Threads:

- Main-Thread  
Responsible for starting the other Threads
- UART-In-Thread  
Responsible for incoming UART-Traffic and the Statemachine
- Processing Thread  
Responsible for Crypto-Operations
- UART-Out-Thread  
Responsible for outgoing UART-Traffic

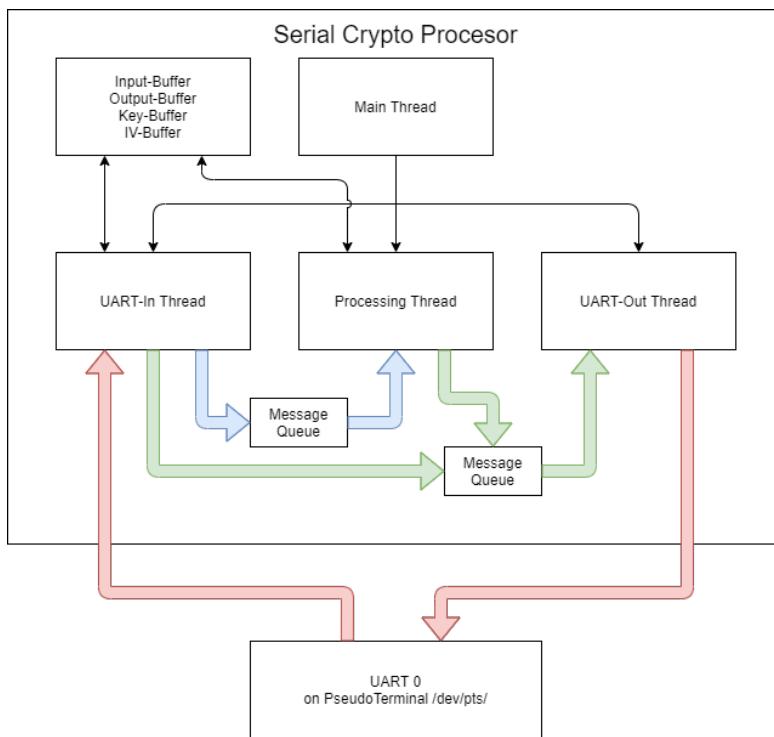


Figure 4: Serial Crypto Processor Block Diagram

## 4.2 Build-Settings

```

1 cmake_minimum_required(VERSION 3.13.1)
2
3 project(CRYPTO_UART)
4 find_package(Zephyr REQUIRED HINTS ${ENV{ZEPHYR_BASE}})
5
6 # Get all ".c"-Files in the src-Directory
7 FILE(GLOB MyCSources src/*.c)
8 target_sources(app PRIVATE ${MyCSources})

```

Listing 1: CMakelists.txt

Get Dependencies for the Serial UART-Device and the Crypto-Device according to [Zephyr KConfig Documentation](#)

```

1 # General config
2 CONFIG_NEWLIB_LIBC_NANO=n
3
4 # Configure Serial-Connection
5 CONFIG_SERIAL=y
6 CONFIG_UART_NATIVE_POSIX=y
7 CONFIG_NATIVE_UART_0_ON_own_PTY=y
8
9 # Configure Crypto Device Dependencies
10 CONFIG_CRYPTO=y
11 CONFIG_TINYCRYPT=y
12 CONFIG_TINYCRYPT_AES_CBC=y
13 CONFIG_CRYPTO_TINYCRYPT_SHIM=y

```

Listing 2: prj.conf

```

1 # This makefile builds the sample for a POSIX system, like Linux
2
3 eventfd: src/main.c
4     $(CC) $^ -o $@

```

Listing 3: Makefile posix

## 4.3 Initialisation

Initialisation consists of:

1. Message-Queue Initialisation
2. UART\_0 Initialisation
3. Crypto-Device Initialisation
4. Thread Initialisation

### 4.3.1 Message-Queue Initialisation

The Message-Queues can be initialised using a Macro which is already defined by Zephyr :

```
1 K_MSGQ_DEFINE(<Name>, <Data-Size>, <Queue-Length>, <Queue-Alignment>);
```

Listing 4: Message Queue Initialisation

During Initialisation 2 Message-Queues are defined, one for the UART-Out-Thread and one for the Processing-Thread:

```
1 // Create Message-Queues using Macros created by Zephyr
2 K_MSGQ_DEFINE(message_queue, sizeof(struct uart_message *), 20, 1);
3 K_MSGQ_DEFINE(crypto_queue, sizeof(char *), 20, 1);
```

### 4.3.2 UART\_0 Initialisation

The Initialisation of the UART\_0-Device for the native\_posix-Board solely consists of getting the Handle for it :

```
1 // Get Handle to the UART_0-Device
2 uart_dev = device_get_binding(UART_DRV_NAME);
3
4 // Check that the UART_0-Device-Handle is correct
5 if (!uart_dev) {
6     return -1;
7 }
```

Listing 5: UART 0 Initialisation

On the native\_posix-Board the Configuration of the Connection Parameters for the UART can be ignored because Pseudo-Terminal are basically Buffers.

On other Boards the correct Connection Parameters need to be set, otherwise the Communication would not happen correctly:

```
1 // Create UART_Config
2 const struct uart_config uart_cfg = {
3     .baudrate = 115200,
4     .parity = UART_CFG_PARITY_NONE,
5     .stop_bits = UART_CFG_STOP_BITS_1,
6     .data_bits = UART_CFG_DATA_BITS_8,
7     .flow_ctrl = UART_CFG_FLOW_CTRL_NONE
8 };
9
10 // Configure UART_0—Device
11 if(!uart_configure(uart_dev, &uart_cfg)) {
12     return -1;
13 }
```

Listing 6: UART-0 Configuration

### 4.3.3 Crypto-Device Initialisation

The Initialisation of the Crypto-Device consists of getting the Device-Handle and ensuring that the Hardware supports the needed Capabilities for the Device:

```

1 // Get Handle to Crypto_Device
2 crypto_dev = device_get_binding(CRYPTO_DRV_NAME);
3
4 // Check that the Crypto—Device—Handle is correct
5 if (!crypto_dev) {
6     return -1;
7 }
8
9 // Ensure that the Crypto—Device has the neccessary Hardware
10 if (validate_hw_compatibility(crypto_dev)) {
11     return -1;
12 }
```

Listing 7: Crypto-Device Initialisation

```

1 // Create global Struct for Crypto—Hardware—Capability—Flags
2 static uint32_t cap_flags;
3
4 // Ensure that the Device has the Capabilities to encrypt using TinyCrypt
5 int validate_hw_compatibility(const struct device *dev) {
6
7     uint32_t flags = cipher_query_hwcaps(dev);
8     if ((flags & CAP_RAW_KEY) == 0U) {
9         return -1;
10    }
11    if ((flags & CAP_SYNC_OPS) == 0U) {
12        return -1;
13    }
14    if ((flags & CAP_SEPARATE_IO_BUFS) == 0U) {
15        return -1;
16    }
17    cap_flags = CAP_RAW_KEY | CAP_SYNC_OPS | CAP_SEPARATE_IO_BUFS;
18    return 0;
19 }
```

Listing 8: Crypto Hardware Capability Check

#### 4.3.4 Thread Initialisation

Thread Initialisation happens in the Main-Thread. The Threads are started one after another and their Handles are stored in an Array:

```

1 // Create Array for Thread-Handles
2 pthread_t threads[NUM_THREADS];
3
4 int ret, i;
5 pthread_attr_t attr[NUM_THREADS] = {};
6 void *(*thread_routines[])(void *) = {uart_in_thread,uart_out_thread,process_thread};
7
8 for (i = 0; i < NUM_THREADS; i++) {
9     ret = pthread_create(&threads[i], &attr[i], thread_routines[i], INT_TO_POINTER(i));
10    if (ret != 0) {
11        return -1;
12    }
13 }
```

Listing 9: Thread Initialisation

#### 4.4 UART-In-Thread

The UART-In-Thread works as the Brain of the whole Processor and is build like a State-Machine:

```

1 // Declare Enums for State Machine
2 enum states{
3     ST_INIT,ST_BUSY,ST_AVAIL,ST_ENCRYPT,ST_DECRYPT,
4     ST_DLEN,ST_DATA,ST_KEY,ST_IV,ST_OP_SEL,ST_OP_KEY,
5     ST_OP_IV,ST_OP_DECRYPT,ST_OP_ENCRYPT
6 };
7 enum operations{OP_INIT,OP_KEY,OP_IV,OP_ENCRYPT,OP_DECRYPT};
```

Listing 10: State Definitions

Upon Initialisation the Processor starts in the Initialisation-State which is basically the IDLE-State:

```

1 // Init Program States
2 static enum states prog_state = ST_INIT;
3 volatile static enum states processing_thread_state = ST_INIT;
4 static enum operations prog_operation = OP_INIT;
```

From there on out the UART-In-Thread receives the Serial-Data and handles them accordingly (Pseudo-Code) :

```
1 // Run until the Stop-Flag is set
2 while (!stop_flag) {
3     switch (prog_state) {
4         case ST_INIT:
5             // Wait for incoming Traffic
6             if(!uart_poll_in(uart_dev,&uart_in)){
7                 // Handle Read
8             }
9             case ST_DATA:
10                // Read Data and set Buffer
11                case ST_IV:
12                    // Override IV with Buffer
13                    case ST_KEY:
14                        // Override Key with Buffer
15                        case ST_DECRYPT:
16                            // Decrypt Data and send it via the UART
17                            case ST_ENCRYPT:
18                                // Encrypt Data and send it via the UART
19                                default:
20                                    // Reset Program State
21    }
22 }
```

Listing 11: State Machine Pseudo-Code

## 4.5 UART-Out-Thread

The UART-Out-Thread handles the Processors Serial-Output and ensures that each Message is sent one after another.

For this a Struct was created so the data can be sent in a universal Manner using a Pointer to a Sequence of Characters and the Length of the String which lies at the Pointer's Location:

```

1 struct uart_message{
2     unsigned char * message;
3     uint32_t len;
4 };

```

Listing 12: Message Struct Definition

The UART-Out-Thread basically sits in an Endless-Loop and waits for Messages to come in:

```

1 void * uart_out_thread(void * x) {
2
3     int iLauf = 0;
4     struct uart_message * message;
5
6     // Run until the Stop-Flag is set
7     while (!stop_flag) {
8         // Block until Data is available
9         if(!k_msgq_get(&message_queue,&message,K_NO_WAIT)) {
10             // Send out Message Char by Char
11             while(iLauf < (message->len)) {
12                 uart_poll_out(uart_dev,message->message[iLauf++]);
13             }
14             // Reset Counter
15             iLauf = 0;
16         }
17     }
18     return x;
19 }

```

Listing 13: UART Out Thread Pseudo-Code

## 4.6 Processing Thread

The Processing Thread is the Worker of the Processor and handles the Encryption and Decryption. It is controlled using Commands which are send via a Message-Queue :

```

1 unsigned char * message;
2 // Run until the Stop-Flag is set
3 while (!stop_flag) {
4     if(!k_msgq_get(&crypto_queue,&message,K_NO_WAIT)) {
5         switch (message[0]) {
6             case ENCRYPT_CHAR:
7                 // Encrypt Buffer and send via UART
8             case DECRYPT_CHAR:
9                 // Decrypt Buffer and send via UART
10            case PROCESSING_CHAR:
11                // Send "Processing Available"
12            case WAIT_CHAR:
13                sleep(10);
14            default:
15                break;
16        }
17    }
18 }
```

Listing 14: Processing Thread Pseudo-Code

## 4.7 Encryption and Decryption

### 4.7.1 Input and Output Buffers

The Parameters for Encryption and Decrytion are stored in static global Buffers so they can be accessed by all Threads when neccessary.

```

1 static uint8_t * g_in_buffer;
2 static uint8_t * g_out_buffer;
3 // Create global Buffer-Length-Variable
4 static uint16_t buffer_length;
5
6 // Create contiguous IV and Key with Default-Values "BBBBBBBBBBBBBBBB"
7 // This is Pseudo-Code
8 static uint8_t g_iv_key[AES_IV_LEN + AES_KEY_LEN];
9 memset(g_iv_key, 'B', AES_IV_LEN + AES_KEY_LEN);
```

Listing 15: Global Buffers for Encryption and Decryption

To prevent Race-Conditions with these Buffers the UART-In-Thread can only access the Buffers when the Processing-Thread is not using them.

#### 4.7.2 Implmentation

Encryption and Decryption are implemented in the same Function. The caller can choose whether to encrypt or decrypt using the “en\_decrypt“-Argument:

```

1 uint32_t cbc_mode(const struct device *dev, uint8_t en_decrypt) {
2
3     // Initialise Crypto Context using Key and Hardware Flags
4     struct cipher_ctx ini = {
5         .keylen = AES_KEY_LEN,
6         .key.bit_stream = g_key,
7         .flags = cap_flags,
8     };
9     // Create Buffers
10    struct cipher_pkt buffers = {
11        .in_buf = g_in_buffer,
12        .in_len = in_buffer_len,
13        .out_buf_max = out_buffer_len,
14        .out_buf = g_out_buffer,
15    };
16
17    // Create Cipher Session which could be reused
18    cipher_begin_session(crypto_dev,&ini,en_decrypt);
19
20    // Execute Cipher Operation
21    cipher_cbc_op(&ini,&buffers,);
22
23    // Free Crypto Context
24    cipher_free_session(dev, &ini);
25 }
```

Listing 16: Encryption and Decryption Pseudo-Code

After Encryption and Decryption the Processing-Thread sends the Output-Buffer to the UART-Out-Thread so it can be sent back via UART\_0. Once the Output-Buffer is sent, the Processing-Thread forfeits access of the Buffers so the UART-In-Thread can also modify them.

## 4.8 Unit-Tests

To verify the functionality of the Crypto-Processor a simple Python Script was provided.

In this Python-File 6 Unit-Tests were defined:

- If the Connection via UART works
- If the Crypto-Processor is available
- What is sent when the Crypto-Processor is busy
- What happens if someone tries to decode a faulty dataset
- A Test if a dataset is decrypted correctly using the standard Parameters
- A Decryption-Test using a custom Key and IV

All 6 Unit-Tests were passed by the Crypto-Processor.