

Serial Crypto Processor Using Zephyr

David Rieser
Supervisor : Prof. Lezuo

March 13, 2021

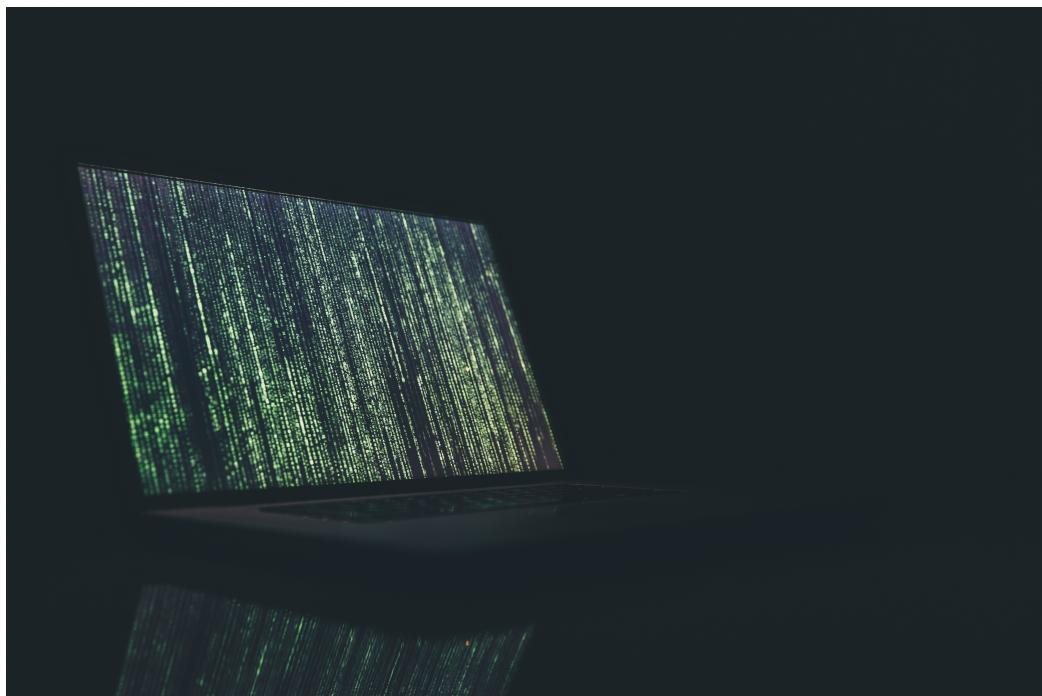


Figure 1: Photo by Markus Spiske on Unsplash

Contents

1	Project Requirements	1
2	Used Technologies	2
2.1	Zephyr	2
2.1.1	KConfig	2
2.1.2	Device Tree	2
2.1.3	Threads	2
2.1.4	Message-Queues	2
2.2	West	3
2.2.1	Ninja	3
2.2.2	CMake	3
2.3	Linux Pseudo-Terminals	3
2.4	TinyCrypt	3
3	Project Execution	4
3.1	Block-Diagram	4
3.2	Build-Settings	5
3.3	Initialisation	6
3.3.1	Message-Queue Initialisation	6
3.3.2	UART_0 Initialisation	6
3.3.3	Crypto-Device Initialisation	8
3.3.4	Thread Initialisation	9
3.4	UART-In-Thread	9
3.5	UART-Out-Thread	10
3.6	Processing Thread	12
3.7	Encryption and Decryption	12

Figures

Statemachine	1
Message-Queue Example Diagram	2
Serial Crypto Processor Block Diagram	4

Listings

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

1 Project Requirements

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

The Processor was 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 or Windows).

The [native_posix-Board](#) has one big limitation which is that Interrupts cannot be implemented. This means the Program had to be developed using non-blocking Calls.

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

The Cryptographic Operations (AES-128 Encryption and Decryption) are implemented using the TinyCrypt-Crypto-Device.

The Crypto-Processor should run using a predefined StateMachine:

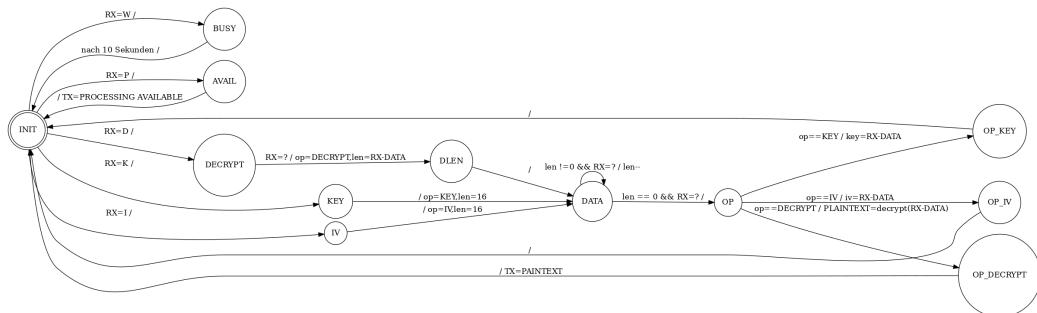


Figure 2: Statemachine

2 Used Technologies

2.1 Zephyr

Zephyr is a small real-time operating system for embedded devices with constraints on hardware resources which supports multiple different architectures developed by [The Linux Foundation](#).

Zephyr is 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](#)

2.1.1 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.

2.1.2 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.

2.1.3 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 simultaniously.

2.1.4 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.

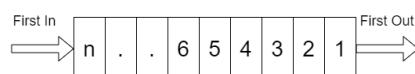


Figure 3: Message-Queue Example Diagram

2.2 West

[West](#) is Zephyr's Meta-Tool. [West](#) initializes, maintains and builds Zephyr-Workspaces using [Ninja](#) and [CMake](#).

2.2.1 Ninja

[Ninja](#) is a small build system that focusses on Speed.

2.2.2 CMake

[CMake](#) is a Tool to automate Building, Testing and Packaging in Software Projects.

It can also generate Ninja Files for faster Building.

2.3 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/.

2.4 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](#).

3 Project Execution

3.1 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

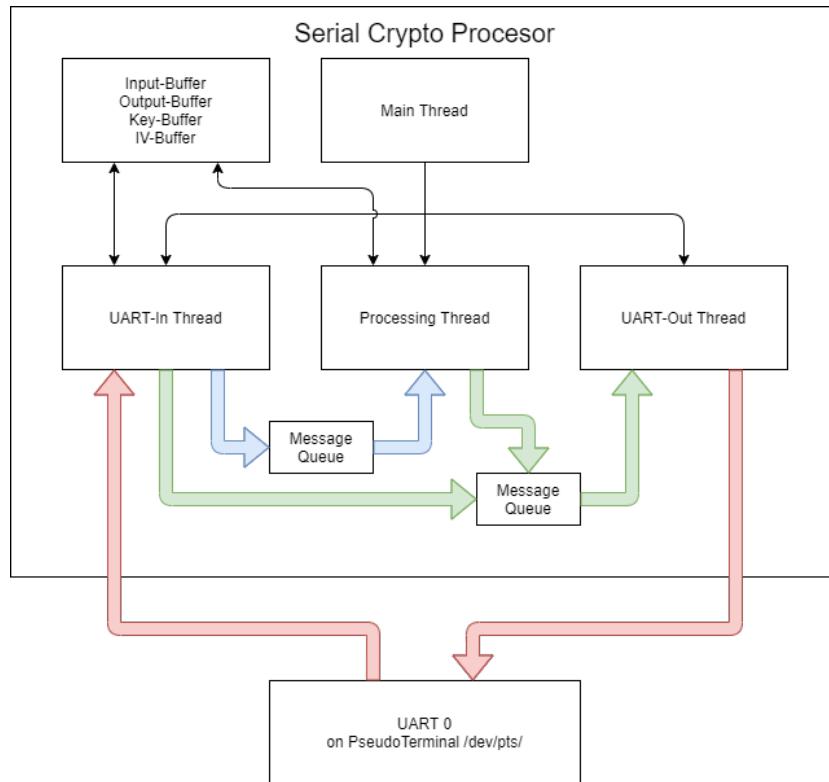


Figure 4: Serial Crypto Processor Block Diagram

3.2 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

3.3 Initialisation

Initialisation consists of:

1. Message-Queue Initialisation
2. UART_0 Initialisation
3. Crypto-Device Initialisation
4. Thread Initialisation

3.3.1 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-Timeout>);
```

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);
```

3.3.2 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

3.3.3 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

3.3.4 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

3.4 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

3.5 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

3.6 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

3.7 Encryption and Decryption

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 static uint8_t g_iv_key[AES_IV_LEN + AES_KEY_LEN] = {
8     0x42, 0x42, 0x42, 0x42, 0x42, 0x42, 0x42, 0x42,
9     0x42, 0x42, 0x42, 0x42, 0x42, 0x42, 0x42,
10    0x42, 0x42, 0x42, 0x42, 0x42, 0x42, 0x42,
11    0x42, 0x42, 0x42, 0x42, 0x42, 0x42, 0x42
12};
```

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.

Encryption and Decryption is implemented in the same Function as there is only one Parameter which is different:

```

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.