Serial Crypto Processor Using Zephyr

David Rieser
Supervisor : Prof. Lezuo

March 18, 2021

Figure 1: Photo by Markus Spiske on Unsplash

Contents

Figures

Listings

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.

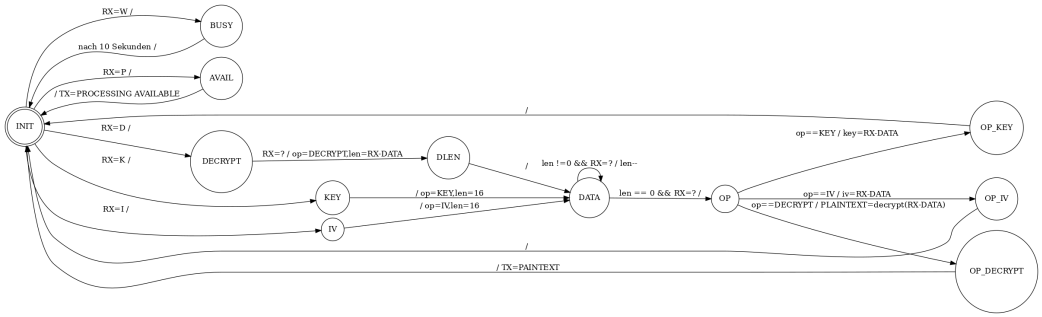

Figure 2: Statemachine

2 Project Implementation Details

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.

2.1 West

West builds very efficiently because it especially targets devices with less resources available.
Because this project uses West, additional functionality can simply be added by changing the Configuration-File.
West also removes the need for a complex build-system but comes with the disadvantage that a big workspace is needed and that a lot of overhead is generated during Compilation.

2.2 Tinycrypt

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 scalabilty if no additional Hardware is provided.

2.3 Program Implementation

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 Used Technologies

3.1 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

3.1.1 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.

3.1.2 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.

3.1.3 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.

3.1.4 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.

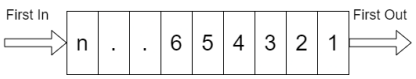

Figure 3: Message-Queue Example Diagram

3.2 West

West is Zephyr's Meta-Tool, that means it uses other tools which in turn build the actual application. West initializes, maintains and builds Zephyr-Workspaces using Ninja and CMake.

3.2.1 CMake

CMake is a Tool to automate Building, Testing and Packaging in Software Projects.
It can also generate Ninja Files for faster Building.

3.2.2 Ninja

Ninja is a small build system that focusses on Speed.

3.3 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/.

3.4 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 Technlolgy Center.

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

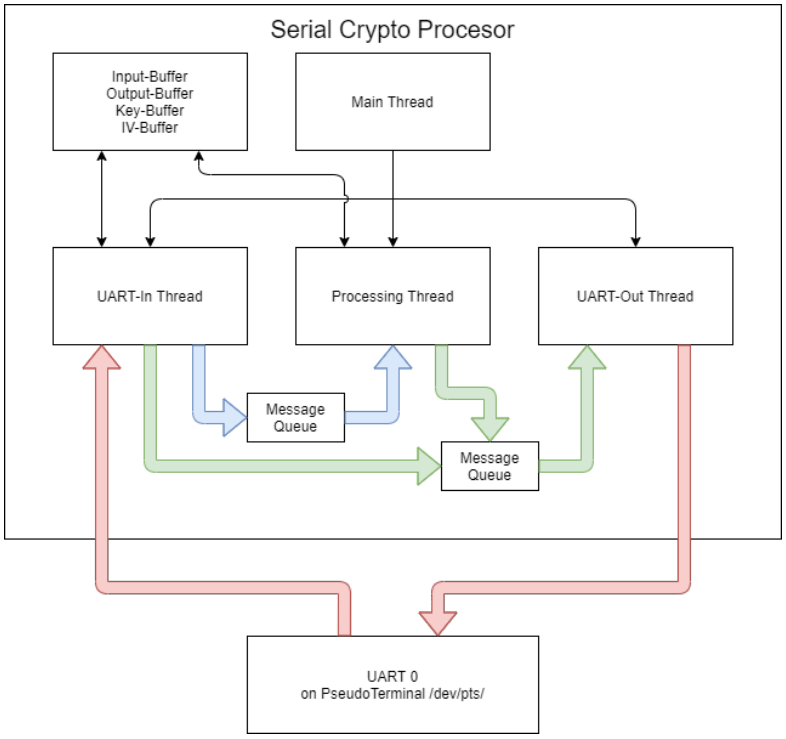

Figure 4: Serial Crypto Processor Block Diagram

4.2 Build-Settings

```
cmake_minimum_required(VERSION 3.13.1)

project(CRYPTO_UART)
find_package(Zephyr REQUIRED HINTS $ENV{ZEPHYR_BASE})

# Get all ".c"-Files in the src-Directory
FILE(GLOB MyCSources src/*.c)
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

```
# General config
CONFIG_NEWLIB_LIBC_NANO=n

# Configure Serial-Connection
CONFIG_SERIAL=y
CONFIG_UART_NATIVE_POSIX=y
CONFIG_NATIVE_UART_0_ON_OWN_PTY=y

# Configure Crypto Device Dependencies
CONFIG_CRYPTO=y
CONFIG_TINYCRYPT=y
CONFIG_TINYCRYPT_AES_CBC=y
CONFIG_CRYPTO_TINYCRYPT_SHIM=y
```

Listing 2: prj.conf

```
# This makefile builds the sample for a POSIX system, like Linux

eventfd: src/main.c
	$(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 :

```
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:

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

```
// Get Handle to the UART_0-Device
uart_dev = device_get_binding(UART_DRV_NAME);

// Check that the UART_0-Device-Handle is correct
if (!uart_dev) {
  return -1;
}
```

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:

```
// Create UART_Config
const struct uart_config uart_cfg = {
  .baudrate = 115200,
  .parity = UART_CFG_PARITY_NONE,
  .stop_bits = UART_CFG_STOP_BITS_1,
  .data_bits = UART_CFG_DATA_BITS_8,
  .flow_ctrl = UART_CFG_FLOW_CTRL_NONE
};

// Configure UART_0-Device
if(!uart_configure(uart_dev, &uart_cfg)) {
  return -1;
}
```

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:

```
// Get Handle to Crypto_Device
crypto_dev = device_get_binding(CRYPTO_DRV_NAME);

// Check that the Crypto-Device-Handle is correct
if (!crypto_dev) {
  return -1;
}

// Ensure that the Crypto-Device has the neccessary Hardware
if (validate_hw_compatibility(crypto_dev)) {
  return -1;
}
```

Listing 7: Crypto-Device Initialisation

```
// Create global Struct for Crypto-Hardware-Capability-Flags
static uint32_t cap_flags;

// Ensure that the Device has the Capabilities to encrypt using Tinycrypt
int validate_hw_compatibility(const struct device *dev) {

    uint32_t flags = cipher_query_hwcaps(dev);
    if ((flags & CAP_RAW_KEY) == 0U) {
        return -1;
    }
    if ((flags & CAP_SYNC_OPS) == 0U) {
        return -1;
    }
    if ((flags & CAP_SEPARATE_IO_BUFS) == 0U) {
        return -1;
    }
    cap_flags = CAP_RAW_KEY | CAP_SYNC_OPS | CAP_SEPARATE_IO_BUFS;
  return 0;
}
```

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:

```
// Create Array for Thread-Handles
pthread_t threads[NUM_THREADS];

int ret, i;
pthread_attr_t attr[NUM_THREADS] = {};
void *(*thread_routines[])(void *) = {uart_in_thread,uart_out_thread,process_thread};

for (i = 0; i < NUM_THREADS; i++) {
  ret = pthread_create(&threads[i], &attr[i], thread_routines[i], INT_TO_POINTER(i));
  if (ret != 0) {
    return -1;
  }
}
```

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:

```
// Declare Enums for State Machine
enum states{
  ST_INIT,ST_BUSY,ST_AVAIL,ST_ENCRYPT,ST_DECRYPT,
  ST_DLEN,ST_DATA,ST_KEY,ST_IV,ST_OP_SEL,ST_OP_KEY,
  ST_OP_IV,ST_OP_DECRYPT,ST_OP_ENCRYPT
};
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:

```
// Init Program States
static enum states prog_state = ST_INIT;
volatile static enum states processing_thread_state = ST_INIT;
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) :

```
// Run until the Stop-Flag is set
while (!stop_flag) {
  switch (prog_state) {
    case ST_INIT:
            // Wait for incoming Traffic
            if(!uart_poll_in(uart_dev,&uart_in)){
        // Handle Read
      }
      case ST_DATA:
      // Read Data and set Buffer
      case ST_IV:
      // Override IV with Buffer
      case ST_KEY:
      // Override Key with Buffer
      case ST_DECRYPT:
      // Decrypt Data and send it via the UART
      case ST_ENCRYPT:
      // Encrypt Data and send it via the UART
      default:
      // Reset Program State
  }
}
```

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:

```
struct uart_message{
  unsigned char * message;
  uint32_t len;
};
```

Listing 12: Message Struct Definition

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

```
void * uart_out_thread(void * x) {

  int iLauf = 0;
  struct uart_message * message;

  // Run until the Stop-Flag is set
  while (!stop_flag) {
    // Block until Data is available
    if(!k_msgq_get(&message_queue,&message,K_NO_WAIT)) {
      // Send out Message Char by Char
      while(iLauf < (message->len)) {
        uart_poll_out(uart_dev,message->message[iLauf++]);
      }
      // Reset Counter
      iLauf = 0;
    }
  }
  return x;
}
```

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 :

```
unsigned char * message;
// Run until the Stop-Flag is set
while (!stop_flag) {
  if(!k_msgq_get(&crypto_queue,&message,K_NO_WAIT)) {
      switch (message[0]) {
        case ENCRYPT_CHAR:
          // Encrypt Buffer and send via UART
        case DECRYPT_CHAR:
          // Decrypt Buffer and send via UART
        case PROCESSING_CHAR:
          // Send "Processing Available"
        case WAIT_CHAR:
          sleep(10);
        default:
          break;
    }
  }
}
```

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.

```
static uint8_t * g_in_buffer;
static uint8_t * g_out_buffer;
// Create global Buffer-Length-Variable
static uint16_t buffer_length;

// Create contingous IV and Key with Default-Values "BBBBBBBBBBBBBBBB"
// This is Pseudo-Code
static uint8_t g_iv_key[AES_IV_LEN + AES_KEY_LEN];
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 Funtion. The caller can choose wheter to encrypt or decrypt using the “en_decrypt“-Argument:

```
uint32_t cbc_mode(const struct device *dev, uint8_t en_decrypt) {

  // Initialise Crypto Context using Key and Hardware Flags
  struct cipher_ctx ini = {
    .keylen = AES_KEY_LEN,
    .key.bit_stream = g_key,
    .flags = cap_flags,
  };
  // Create Buffers
  struct cipher_pkt buffers = {
    .in_buf = g_in_buffer,
    .in_len = in_buffer_len,
    .out_buf_max = out_buffer_len,
    .out_buf = g_out_buffer,
  };

  // Create Cipher Session which could be reused
  cipher_begin_session(crypto_dev,&ini,en_decrypt);

  // Execute Cipher Operation
  cipher_cbc_op(&ini,&buffers,);

  // Free Crypto Context
  cipher_free_session(dev, &ini);
}
```

Listing 16: Encryption and Decryption Pseudo-Code

After Encryption and Decrytion 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 definded:

- 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.