



CCA – COMPETENCE CENTRE

HTL Anichstraße

DIC-serial_crypto

Fabio Plunser

13. April 2021



Zephyr™

Inhaltsverzeichnis

1	Information	1
2	Aufgabenstellung	1
2.1	Aufgaben und Eigenschaften des Krypto Prozessors	1
3	Theorie und Vorwissen	3
3.1	Zephyr	3
3.1.1	KConfig	3
3.1.2	Device Tree	4
3.1.3	Tinycrypt	5
3.2	Linux Pseudoterminal	5
3.3	Threads	5
3.4	Message-Queue	6
4	Programm Umsetzung	7
4.1	Blockschaltbild	7
4.2	Projekt Konfiguration	8
4.3	Initialisierung	8
4.3.1	Message-Queue	8
4.3.2	UART_0	9
4.3.3	Verschlüsselung	10
4.3.4	Threads	11
4.4	UART-IN-Thread	11
4.5	UART-Out-Thread	13
4.6	Processing Thread	14
4.6.1	Entschlüsselung	15
4.7	Test-Ausführung	16

Abbildungsverzeichnis

1	Statemachine	1
2	Process-Threads	5
3	Message-Queue-Darstellung	6
4	Blockschaltbild	7
5	Test00-01	16
6	Test02	16
7	Test03	16
8	Test04	17
9	Test05	18

Code

1	West Beispiel	3
2	West Beispiel	4
3	prj.conf	8
4	Message-Queue-Initialisierung	8
5	UART-Initialisierung	9
6	UART-Initialisierung	10
7	UART-Initialisierung	11
8	Statemachine-Enumerations	11
9	UART-IN-Thread	11
10	Statemachine	12
11	uart_message-struct label	13
12	UART-OUT-Thread	13
13	Processing-Thread	14
14	UART-Initialisierung	15

1 Information

Vielen dank an David Reiser, der mir sein Makefile zur Verfügung gestellt hat.

Weiterhin kann das komplette Projekt in folgender GITHUB Repository nachvollzogen werden:
<https://github.com/FabioPlunser/DIC-Lezuo/>

2 Aufgabenstellung

Die Aufgabe ist es, im echtzeit Betriebssystem Zephyr einen Krypto Prozessor zu programmieren, der einen Verschlüsselten Text erhält und mit AES-128 cbc entschlüsselt. Der Prozessor wird mit dem **nativ_posix-Board** programmiert. Dieses kann in eine normal ausführbare Datei kompiliert werden, die man auf einem Linux System ausführen kann. Somit wird ein Mikronroller Board emuliert.

2.1 Aufgaben und Eigenschaften des Krypto Prozessors

Der Krypto Prozessor soll in 4 Threads, **main**, **uart-in**, **uart-out**, **processing** aufgeteilt werden. Weiterhin soll die vorgegebene Statemachine und UART Protokoll implementiert werden. Die Statemachine gibt vor in wann das Programm was machen soll.

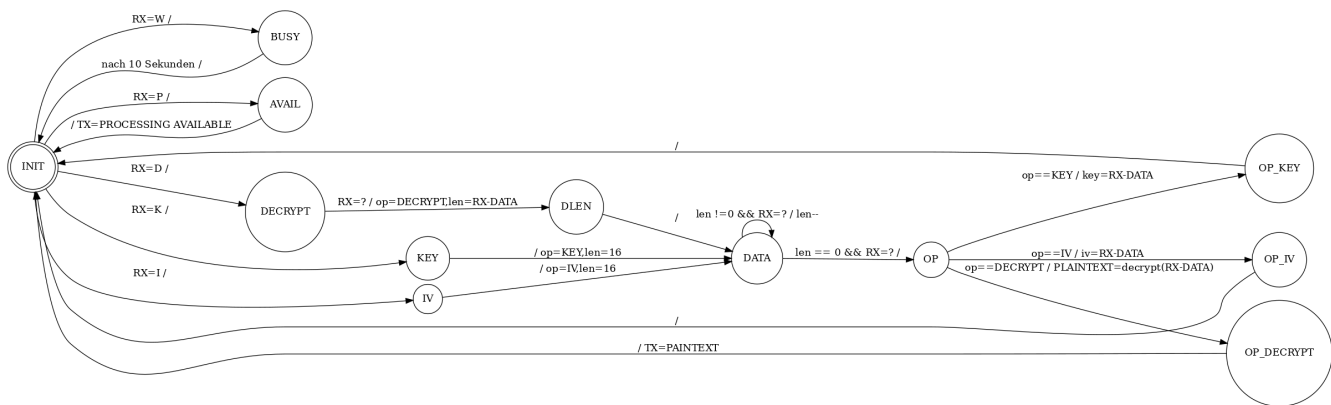


Abbildung 1: Statemachine

Weiterhin wurde ein UART Protokoll vorgegeben:

- alive: Wenn ein ' ' empfangen wird, soll sofort ein ' ' zurückgeschickt werden.
- avail: Wenn ein 'P' empfangen wird, soll vom processing-thread „PROCESSING AVAI“ zurückgeschickt werden.
- key: Wenn ein 'K' empfangen wird, folgen 16 Byte Des AES-128 Schlüssel, dieser empfangene Schlüssel wird in den Kryptoprozessor geladen.
- iv: Wenn ein 'I' empfangen wird, folgen 16 Byte des AES-128 IV, , dieser empfangene IV wird in den Kryptoprozessor geladen.

- Decrypt: Wenn ein 'D' empfangen wird, gefolgt von der Länge des Ciphertextes, gefolgt vom Ciphertext, wird dieser Ciphertext mit dem entsprechenden Key und IV mit AES128-CBC entschlüsselt und als Plaintext an der UART ausgegeben. Wenn der Ciphertext nicht durch 16 Teilbar ist, soll eine Fehlermeldung „XERROR“ zurückgesendet werden.

Das Programm soll alle Tests der vorgegebenen test.py Datei erfolgreich absolvieren. Die Tests, testen ob die Statemachine korrekt implementiert wurde und besteht aus folgende Test:

- Test0: Testung der UART Verbindung, indem ein '.' Punkt an den Prozessor geschickt wird.
- Test1: Testung der availability, indem ein 'P' an den Prozessor geschickt wird.
- Test2: Testung ob der Processor korrekt blockiert
- Test3: Testung ob ein Error vom Prozessor zurückgeschickt wird, wenn ein absichtlich nicht funktionierender Ciphertext an den Prozessor geschickt wird, da dieser nicht durch 16 Teilbar ist.
- Test4: Testung ob die standard Konfiguration der Entschlüsselt korrekt ist.
- Test5: Testung ob ein anderer Key und IV von dem Prozessor übernommen wird.

3 Theorie und Vorwissen

3.1 Zephyr

Zephyr ist ein Open-Source-Echtzeitbetriebssystem welches von der Linux Foundation.¹ Ein Echtzeitbetriebssystem, real-time operating system **RTOS** ist ein Betriebssystem, das Echtzeit-Anforderungen erfüllen kann. Das bedeutet, dass Anfragen eines Anwendungsprogramms innerhalb einer Voraus bestimmbarer Zeit gesichert verarbeitet werden.²

Zephyr wurde mit dem Getting-Started-GUID Linux Subsystem von Windows installiert. Um ein Zephyr Projekt zu kompilieren wird Zephyr eigenes **West**³ verwendet.

West ist ein Kompilierungs-Tool von Zephyr. Es verwendet Ninja und CMake um das Projekt zu kompilieren. West wird folgendermaßen verwendet, um ein Projekt zu kompilieren:

```
1 west build -p auto -b native_posix_64
```

Listing 1: West Beispiel

3.1.1 KConfig

Kernel Configuration File⁴ ist die **prj.conf** Datei in einem Zephyr Projekt. In diesem werden bestimmte Konfigurationen, Funktionen und „Geräte“, wie z.B. *CONFIG_SERIAL=y* aktiviert.

¹Quelle: [https://de.wikipedia.org/wiki/Zephyr_\(Betriebssystem\)](https://de.wikipedia.org/wiki/Zephyr_(Betriebssystem))

²Quelle: <https://de.wikipedia.org/wiki/Echtzeitbetriebssystem>

³<https://docs.zephyrproject.org/2.4.0/guides/west/index.html>

⁴<https://docs.zephyrproject.org/latest/application/index.html?#application-kconfig>

3.1.2 Device Tree

Der Device Tree⁵ ist in einem Zephyr Projekt eine Datei mit der Endung `.dts` dort stehen alle für das ausgewählte Board verfügbare Geräte drinnen. Im Fall des `nativ_posix_64` sieht dieses folgendermaßen aus.

```

1  /dts-v1/;
2
3  / {
4      #address-cells = < 0x1 >;
5      #size-cells = < 0x1 >;
6      model = "Native POSIX Board";
7      compatible = "zephyr,posix";
8      chosen {
9          zephyr,console = &uart0;
10         zephyr,shell-uart = &uart0;
11         zephyr,uart-mcumgr = &uart0;
12         zephyr,flash = &flash0;
13         zephyr,entropy = &rng;
14         zephyr,flash-controller = &flashcontroller0;
15         zephyr,ec-host-interface = &hcp;
16     };
17     aliases {
18         eeprom-0 = &eeprom0;
19         i2c-0 = &i2c0;
20         spi-0 = &spi0;
21         led0 = &led0;
22     };
23     leds {
24         compatible = "gpio-leds";
25         led0: led_0 {
26             gpios = < &gpio0 0x0 0x0 >;
27             label = "Green LED";
28         };
29     };
30     ...
31 };
32
33 uart0: uart {
34     status = "okay";
35     compatible = "zephyr,native-posix-uart";
36     label = "UART_0";
37     current-speed = < 0x0 >;
38 };
39
40 ...
41
42 };
43

```

Listing 2: West Beispiel

⁵<https://docs.zephyrproject.org/latest/guides/dts/intro.html>
<https://docs.zephyrproject.org/latest/reference/devicetree/index.html#devicetree>

3.1.3 Tinycrypt

Die TinyCrypt-Bibliothek bietet eine Implementierung für eingeschränkte Geräte von minimalen Standard-Kryptographie-Grundelementen. Die Bibliothek ist von Intel⁶ und wurde in Zephyr mit der implementiert.⁷ Innerhalb von Zephyr können die eigenen Crypto API Befehle oder die direkten TinyCrypt Befehle verwendet werden.

3.2 Linux Pseudoterminal

Ein Pseudoterminal ist ein Dienst der eine bidirektionale Pipe, aufbaut. Sie werden verwendet um ein physisches Terminal zu emulieren. Im Fall von Zephyr mit dem `nativ_posix` Board wird ein Pseudoterminal verwendet um mit dem Board zu kommunizieren. Der Pfad dieses Terminals ist `/dev/pts/`.

3.3 Threads

Innerhalb eines Betriebssystems werden Applikationen und Abläufe als Prozesse realisiert. Prozesse werden vom Scheduler verwaltet. Um nun die Abarbeitung dieser Prozesse zu beschleunigen, besteht ein Prozess aus mehreren parallel laufenden Threads. Diese Threads beinhalten einen Teil des Prozesses, dadurch können große Probleme bestehen. Es muss sichergestellt werden, dass Threads nie auf die gleichen Variablen/Ressourcen zugreifen, da ansonsten Race-Conditions entstehen.

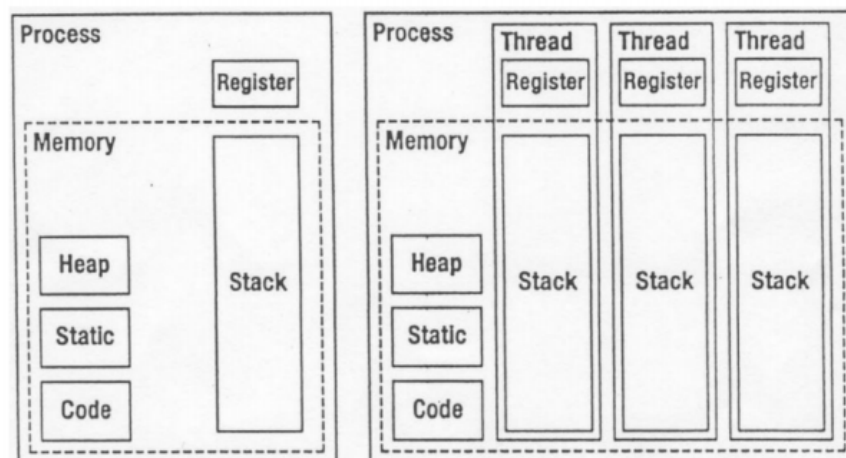


Abbildung 2: Process-Threads

⁶<https://github.com/intel/tinycrypt>

⁷<https://docs.zephyrproject.org/2.3.0/guides/crypto/tinycrypt.html?highlight=tinycrypt>

3.4 Message-Queue

Eine Message-Queue ist ein besonderer Buffer, ein FIFO-Buffer (First-In-First-Out-Buffer). Das bedeutet, es werden Nachrichten in einer Reihe in den Buffer geschrieben und es kann nur die erste Nachricht in der Reihe herausgenommen werden, dabei wird diese Nachricht im Buffer gelöscht und die nächste Nachricht rückt nach. Solche Message-Queues werden verwendet, um zwischen Threads die Daten korrekt auszutauschen. Da eine Nachricht beim Auslesen gelöscht wird, können nicht mehrere Threads gleichzeitig auf die Nachricht zugreifen.

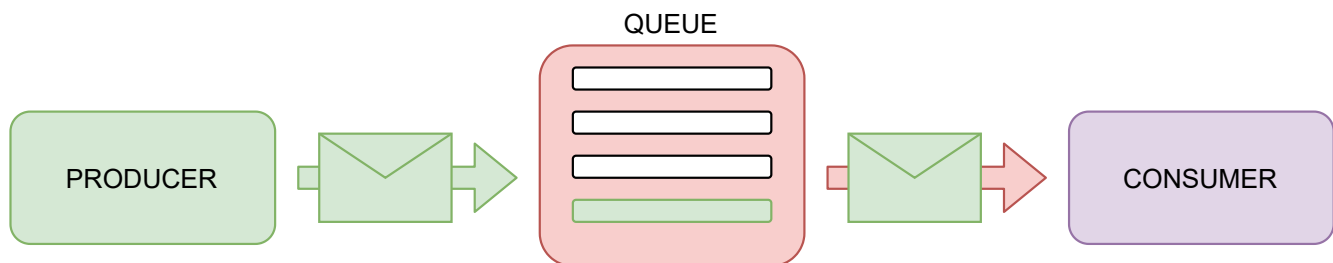


Abbildung 3: Message-Queue-Darstellung

4 Programm Umsetzung

Das Programm wurde wie angegeben in 4 Threads eingeteilt.

- Main-Thread
 - Initialisierung der anderen Threads
 - UART und Crypto Device Initialisierung
 - Validate Hardware Compatibility
 - alle 5 Sekunden ein Lebenszeichen von sich geben.
- UART-IN-Thread
 - Einlesen der UART
 - Statemachine Implementierung
- UART-OUT-Thread
 - Ausgabe der in die Queue geschriebenen Messages
- PROCESS-Thread
 - Verwaltung der Entschlüsselung

4.1 Blockschaftbild

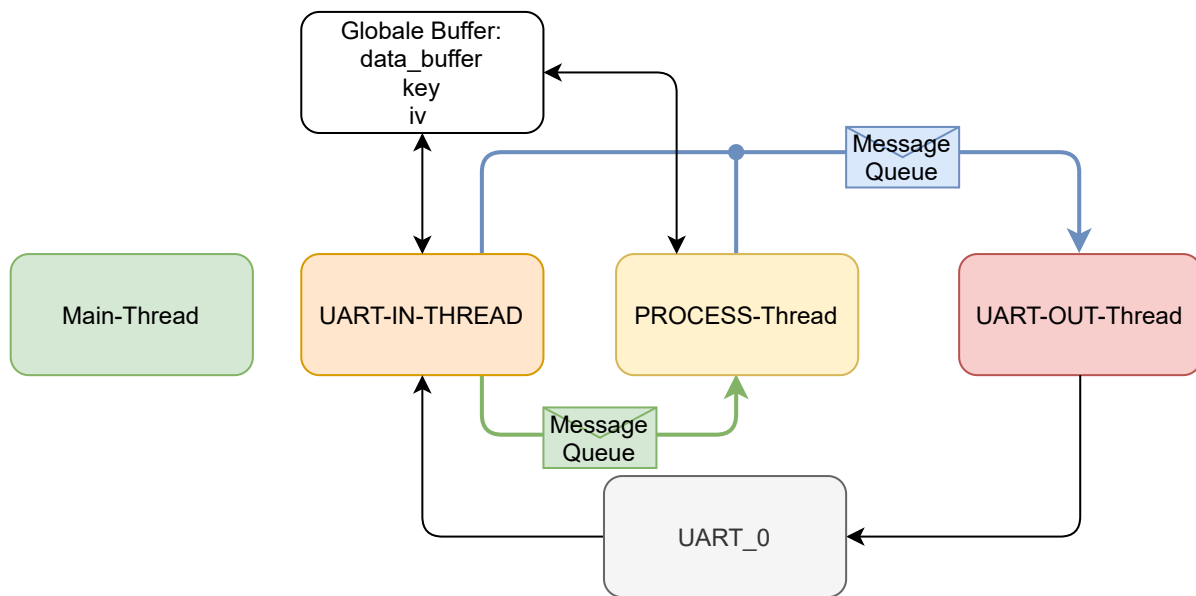


Abbildung 4: Blockschaftbild

4.2 Projekt Konfiguration

Sodass, das `nativ_posix` Board die benötigten Geräte verwenden kann müssen dieser in der `prj.conf` Datei aktivieren werden.

```
1 #Configure Serial-Connection
2 CONFIG_SERIAL=y
3 CONFIG_UART_NATIVE_POSIX=y
4 CONFIG_NATIVE_UART_0_ON_OWN_PTY=y
5
6 #Crypto
7 CONFIG_TINYCRYPT=y
8 CONFIG_TINYCRYPT_AES=y
9 CONFIG_TINYCRYPT_AES_CBC=y
10 CONFIG_CRYPT=y
11 CONFIG_CRYPT_TINYCRYPT_SHIM=y
```

Listing 3: prj.conf

4.3 Initialisierung

Es müssen:

- Message-Queue
- UART_0
- Verschlüsselung
- Threads

initialisiert werden.

4.3.1 Message-Queue

Wie in der Zephyr Dokumentation⁸ beschrieben wird, kann eine Message-Queue mit einem Macro initialisiert werden. Für den Kryptoprozessor werden zwei Message-Queues verwendet, eine um die Nachrichten an der UART auszugeben und eine weitere um Befehle an den Process-Thread weiterzugeben.

```
1 K_MSGQ_DEFINE(uart_queue, sizeof(struct uart_message *), q_max_msgs, q_align);
2 K_MSGQ_DEFINE(crypto_queue, sizeof(char* ), q_max_msgs, q_align);
```

Listing 4: Message-Queue-Initialisierung

⁸https://docs.zephyrproject.org/2.3.0/reference/kernel/data_passing/message_queues.html?highlight=queue

4.3.2 UART_0

Um die UART verwenden zu können muss zuerst ein Device „erstellt“ werden. Dieses Gerät muss dann zur richtigen UART_0 *gebinded*/verbunden werden. Danach kann die UART wie in der Dokumentation⁹ beschrieben konfiguriert werden. Da die UART in diesem Fall nur mit Pseudo-Terminal verwendet wird, ist die Konfiguration der Baudrate etc. nicht notwendig.

```
1  const struct device * uart_dev;  
2  uart_dev = device_get_binding(UART_NAME);  
3  if(!uart_dev){  
4      printk("UART-binding-error\n");  
5  }  
6  const struct uart_config uart_cfg = {  
7      .baudrate = 115200,  
8      .parity = UART_CFG_PARITY_NONE,  
9      .stop_bits = UART_CFG_STOP_BITS_1,  
10     .data_bits = UART_CFG_DATA_BITS_8,  
11     .flow_ctrl = UART_CFG_FLOW_CTRL_NONE  
12 };  
13  
14 if(!uart_configure(uart_dev, &uart_cfg)){  
15     printk("UART-config-error\n");  
16 }
```

Listing 5: UART-Initialisierung

⁹<https://docs.zephyrproject.org/2.3.0/reference/peripherals/uart.html>

4.3.3 Verschlüsselung

Die Initialisierung der Verschlüsselung bzw. des Crypto Device funktioniert sehr ähnlich wie bei der UART. Nur statt eine eigenen Konfiguration wird eine Validate_Hardware_Funktion verwendet um zu Überprüfen wie das Crypto Device verwendet werden kann. Diese Funktion wurde vom CBC Beispiel von Zephyr kopiert.

```
1  const struct device * crypto_dev;
2  static uint32_t cap_flags;
3
4  //bind crypto
5  crypto_dev = device_get_binding(CRYPTO_DRV_NAME);
6  if (!crypto_dev) {
7      printk("Crypto-binding-error\n");
8      return;
9  }
10 //validate hardware for crypto device
11 validate_hw_compatibility();
12
13
14 int validate_hw_compatibility()
15 {
16     uint32_t flags = 0U;
17     flags = cipher_query_hwcaps(crypto_dev);
18     if ((flags & CAP_RAW_KEY) == 0U) {
19         printk("Please provision the key separately "
20             "as the module doesnt support a raw key\n");
21         return -1;
22     }
23
24     if ((flags & CAP_SYNC_OPS) == 0U) {
25         printk("The app assumes sync semantics. "
26             "Please rewrite the app accordingly before proceeding\n");
27         return -1;
28     }
29
30     if ((flags & CAP_SEPARATE_IO_BUFS) == 0U) {
31         printk("The app assumes distinct IO buffers. "
32             "Please rewrite the app accordingly before proceeding\n");
33         return -1;
34     }
35     cap_flags = CAP_RAW_KEY | CAP_SYNC_OPS | CAP_SEPARATE_IO_BUFS;
36     return 0;
37 }
```

Listing 6: UART-Initialisierung

4.3.4 Threads

Die Threads werden innerhalb des Main-Thread initialisiert. Dabei sind die Threads in einem Array und werden nacheinander gestartet.

```
1 pthread_t thread_id[Number_of_threads];
2 init_threads(thread_id);
3
4 void init_threads(pthread_t* thread_id)
5 {
6     //init threads
7     int i, thread_ok;
8     void*(*threads[]) (void*) = {uart_in_thread, uart_out_thread,
9         process_thread};
10    for(i=0; i<Number_of_threads; i++)
11    {
12        thread_ok = pthread_create(&thread_id[i], NULL, threads[i], NULL);
13        if(thread_ok != 0)
14        {
15            printf("Thread creation Error\n");
16        }
17    }
```

Listing 7: UART-Initialisierung

4.4 UART-IN-Thread

Im UART-IN-Thread ist die Statemachine implementiert. Somit steuert dieser Thread alle Vorgänge im Prozessor. Die States werden mittels einer Switch-Case abgefragt und gesetzt. Um die States übersichtlich zu setzen wurde eine Enumeration verwendet.

```
1 enum state{ INIT, ALIVE, AVAIL, KEY, IV, DECRYPT, DLEN, DATA,
2     SELECT_OPERATION};
3 enum op{OP_KEY, OP_IV, OP_DECRYPT};
```

Listing 8: Statemachine-Enumerations

```
1 void* uart_in_thread(void * x){
2     state_machine();
3     return x;
4 }
```

Listing 9: UART-IN-Thread

```

1 void state_machine()
2 {
3     uint8_t i = 0;
4     uint8_t uart_in;
5     uint8_t* data_buffer = "";
6     printk("In State Machine\n");
7     while(1)
8     {
9         switch(st_state){
10             case INIT:
11                 if(!uart_poll_in(uart_dev, &uart_in)){
12                     switch(uart_in){
13                         case 'w':
14                         case 'W':
15                             put_message_in_crypto_queue("W\n");
16                             break;
17                         ...
18                     }}
19                 break;
20             ...
21             Pseudo-Code
22             case DECRYPT
23                 //set busy flag, set op=OP_KEY, goto DLEN
24             case IV
25                 //allocate data_buffer, set op=OP_IV, goto DATA
26             case KEY
27                 //allocate data_buffer, set op=OP_KEY, goto DATA
28             case DLEN
29                 //allocate data_buffer for data + iv, copy iv into buffer, move
30                 //pointer from buffer where data should start, goto data
31             CASE DATA
32                 //get data from uart and put it into data_buffer, goto
33                 SELECT_OPERATION
34             case SELECT_OPERATION:
35                 switch(operation)
36                 {
37                     case OP_KEY:
38                         //copy key from data_buffer into global key variable,
39                         //goto state = INIT
40                     case OP_IV:
41                         //copy iv from data_buffer into global iv variable, goto
42                         //state = INIT
43                     case OP_DECRYPT
44                         //reset pointer from data_buffer, copy data_buffer into
45                         //a global buffer, goto state = INIT
46                 }
47             }
48         }
49     }
50 }

```

Listing 10: Statemachine

4.5 UART-Out-Thread

Der UART-OUT-Thread sendet die Daten, die in der UART-Message-Queue stehen. Da die Daten für die finalen Tests teilweise mit Nullterminierung geschickt werden müssen wurde für die UART-Message-Queue ein Struct erstellt um die länge der Message festzulegen, da *strlen()* nur bis zur Nullterminierung zählt.

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

Listing 11: uart_message-struct label

```
1 //put string into uart queue
2 int put_message_in_uart_queue(unsigned char* str, uint32_t len)
3 {
4     static struct uart_message message;
5     message.message = str;
6     message.len = len;
7     struct uart_message * message_pointer = &message;
8
9     if(k_msgq_put(&uart_queue, &message_pointer, K_FOREVER)!=0){
10         printk("Couldn't put message in queue!!\n");
11     }
12     return 0;
13 }
14 //send uart messages from queue
15 void* uart_out_thread(void * x)
16 {
17     int i=0;
18     uint32_t len;
19     struct uart_message * message;
20     unsigned char* message_temp;
21     while(1)
22     {
23         if(!k_msgq_get(&uart_queue, &message, K_NO_WAIT)) {
24             len = message->len;
25             message_temp = message->message;
26             while(i < len)
27             {
28                 uart_poll_out(uart_dev, message_temp[i++]);
29             }
30             i = 0;
31         }
32     }
33     return x;
34 }
```

Listing 12: UART-OUT-Thread

4.6 Processing Thread

Der Processing-Thread startet die Entschlüsselung und verwaltet, wie in der Angabe beschrieben, das Senden vom "Processing-Available" und Blocken des Threads.

```
1 void * process_thread(void * x)
2 {
3     unsigned char* message;
4     while(1)
5     {
6         if(!k_msgq_get(&crypto_queue, &message, K_NO_WAIT)) {
7
8             switch (message[0])
9             {
10                case 'D':
11                    printk("Process_thread: Decrypting\n");
12                    printk("Ciphertext: %02X\n", cbc_buffer);
13                    processing_busy = true;
14                    if(cbc_mode())
15                    {
16                        format_plaintext_for_comparison(out_buffer);
17                    }
18                    processing_busy = false;
19                    break;
20                case 'P':
21                    if(processing_busy == false){
22                        put_message_in_uart_queue("PROCESSING AVAILABLE\n", strlen("
23                            PROCESSING AVAILABLE\n"));
24                    }
25                    break;
26                case 'W':
27                    processing_busy = true;
28                    sleep(10);
29                    processing_busy = false;
30                    break;
31
32                default:
33                    break;
34            }
35        }
36    }
37    return x;
38 }
```

Listing 13: Processing-Thread

4.6.1 Entschlüsselung

Für die Entschlüsselung sind standard IV und Key und ein Input und Output Buffer nötig. Diese werden global und in der Statemachine entsprechend der Angabe gesetzt.

```

1 //set default ciphertext BBBBBBBBBBBBBBBB
2 static uint8_t iv[AES_IV_LEN] ={
3     0x42,0x42,0x42,0x42,
4     0x42,0x42,0x42,0x42,
5     0x42,0x42,0x42,0x42,
6     0x42,0x42,0x42,0x42,
7 };
8 //set default key, with care so that iv and key are not overwriting each
  other
9 uint8_t* key = iv;
10 static uint8_t* cbc_buffer;
11 static uint8_t* out_buffer;
12
13 int cbc_mode()
14 {
15     uint32_t in_buffer_len = len + AES_IV_LEN;
16     uint32_t out_buffer_len = len;
17     out_buffer = malloc(out_buffer_len);
18     struct cipher_ctx ini = {
19         .keylen = AES_KEY_LEN,
20         .key.bit_stream = key,
21         .flags = cap_flags,
22     };
23     struct cipher_pkt decrypt = {
24         .in_buf = cbc_buffer,
25         .in_len = in_buffer_len,
26         .out_buf = out_buffer,
27         .out_buf_max = out_buffer_len,
28     };
29     if(cipher_begin_session(crypto_dev, &ini, CRYPTO_CIPHER_ALGO_AES,
        CRYPTO_CIPHER_MODE_CBC, CRYPTO_CIPHER_OP_DECRYPT)){
30         cipher_free_session(crypto_dev, &ini); put_message_in_uart_queue("XERROR
        \n", strlen("XERROR\n")); return 0;
31     }
32     if (cipher_cbc_op(&ini, &decrypt, cbc_buffer)) {
33         cipher_free_session(crypto_dev, &ini); put_message_in_uart_queue("XERROR
        \n", strlen("XERROR\n")); return 0;
34     }
35     cipher_free_session(crypto_dev, &ini);
36     return 1;
37 }

```

Listing 14: UART-Initialisierung

```
-crypto-Progress> make run
Running Project
UART_0 connected to psonnetty: /dev/ttyS6
*** Booting Zephyr OS build zephyr-v2.5.8-1214-gd9f56fe7e163 ***

In State Machine
Main-Thread is alive
Main-Thread is alive
State Machine UART_IN:
Putting into uart_queue: .

Message in queue: .

State Machine UART_IN:
Putting into uart_queue: .

Message in queue: .

State Machine UART_IN:
Putting into uart_queue: .

Message in queue: .

State Machine UART_IN:
Putting into uart_queue: .

Message in queue: .

State Machine UART_IN:
Putting into uart_queue: .

Message in queue: .

State Machine UART_IN:
Putting into uart_queue: .

Message in queue: .

State Machine UART_IN:
Putting into uart_queue: .

Message in queue: .

State Machine UART_IN:
Putting into uart_queue: .

Message in queue: .

State Machine UART_IN: P
Putting into uart_queue: PROCESSING AVAILABLE

Message in queue: PROCESSING AVAILABLE

Main-Thread is alive
```

```

peppi@Peppi-Dekstop: /mnt/c/Users/fplun/GoogleDrive/Schule/2020_21/DIC/DIC-Lezuo/2.Semester-serial_crypto/DIC
test_00 (C:\_main__MyTests) ... 000000.000 Q-RX reset_input_buffer
000000.000 TX 0000 2E
000000.000 RX 0000 2E 0A
Replyb'.'.\n'
000000.000 TX 0000 2E
000000.001 RX 0000 2E 0A
Replyb'.'.\n'
000000.001 TX 0000 2E
000000.001 RX 0000 2E 0A
Replyb'.'.\n'
000000.001 TX 0000 2E
000000.001 RX 0000 2E 0A
Replyb'.'.\n'
000000.001 TX 0000 2E
000000.001 RX 0000 2E 0A
Replyb'.'.\n'
000000.001 TX 0000 2E
000000.001 RX 0000 2E 0A
Replyb'.'.\n'
000000.001 TX 0000 2E
000000.002 RX 0000 2E 0A
Replyb'.'.\n'
000000.002 TX 0000 2E
000000.002 RX 0000 2E 0A
Replyb'.'.\n'
000000.002 TX 0000 2E
000000.002 RX 0000 2E 0A
Replyb'.'.\n'
000000.002 TX 0000 2E
000000.002 RX 0000 2E 0A
Replyb'.'.\n'
000000.002 TX 0000 2E
000000.003 RX 0000 2E 0A
Replyb'.'.\n'
ok
test_01 (C:\_main__MyTests) ... 000000.000 TX 0000 50
000001.001 RX 0000 50 52 4F 43 45 53 53 49 4E 47 20 41 56 41 49 4C PROCESSING AVAIL
000001.001 RX 0010 41 42 4C 45 0A ABLE.
Replyb'PROCESSING AVAILABLE'\n'
nk.

```

```
Main-Thread is alive
State Machine UART_IN: W
State Machine UART_IN: P
State Machine UART_IN: .
Putting into uart_queue: .

Message in queue: .

Putting into uart_queue: BUSY

Message in queue: BUSY

State Machine UART_IN: .
Putting into uart_queue: .

Message in queue: .

Putting into uart_queue: BUSY

Message in queue: BUSY
```

```

test_02 (_main...MyTests) ... 000000.000 TX 0000 57
000000.000 TX 0000 50
Replay!'
000001.001 TX 0000 2E
000002.003 RX 0000 2E 0A 42 55 53 59 0A
000002.003 TX 0000 2E
000003.004 RX 0000 2E 0A 42 55 53 59 0A
000003.004 TX 0000 2E
000004.005 RX 0000 2E 0A 42 55 53 59 0A
000004.005 TX 0000 2E
000005.006 RX 0000 2E 0A 42 55 53 59 0A
000005.007 TX 0000 2E
000006.008 RX 0000 2E 0A 42 55 53 59 0A
000006.008 TX 0000 2E
000007.009 RX 0000 2E 0A 42 55 53 59 0A
000007.009 TX 0000 2E
000008.010 RX 0000 2E 0A 42 55 53 59 0A
000008.010 TX 0000 2E
000009.011 RX 0000 2E 0A 42 55 53 59 0A
000009.012 TX 0000 2E
000010.012 RX 0000 2E 0A 42 55 53 59 0A 50 52 4F 43 45 53 53 49 4E
000010.012 RX 0010 47 20 41 56 41 49 4C 41 42 4C 45 0A
ok.

```

```
State Machine UART_IN: D
DECRYPT
DATA
Data: 0xAA
Data: 0xE3
OP_DECRYPT
Process_thread: Decrypting
Ciphertext: 4008060
CBC_Mode:
Key: 8888888888888888@Q@dU
IV: 8888888888888888@Q@dU
Buffer: 8888888888888888
Buffer 0x4008060
Keylen: 16 keybittstream: 8888888888888888@Q@dU , flags 50
Putting into uart_queue: XERROR

Message in queue: XERROR
```

```

000010.012 RX 0000 2E 0A 42 55 53 59 0A 50 52 4F 43 45 53 53 49 4E ..BUSY.PROCESSING
000010.012 RX 0010 47 20 41 56 41 49 4C 41 42 4C 45 0A G AVAILABLE.
ok
test_03 (__main__.MyTests) ... Data in Bytes: b'D' Data: 68
000000.000 TX 0000 44 D
Data in Bytes: b'\x02' Data: 2
000000.100 TX 0000 02 .
Data in Bytes: b'\xaa' Data: 170
000000.201 TX 0000 AA .
Data in Bytes: b'\xe3' Data: 227
000000.301 TX 0000 E3 .
Data in Bytes: b'X' Data: 88
000000.402 TX 0000 58 X
000001.503 RX 0000 58 45 52 52 4F 52 0A XERROR.
Replyb'XERROR\n'
ok

```

```

test_04 (__main__.MyTests) ... Data in Bytes: b'D' Data: 68
000000.000 TX 0000 44
Data in Bytes: b' ' Data: 32
000000.100 TX 0000 20
Data in Bytes: b'\xaa' Data: 170
000000.201 TX 0000 AA
Data in Bytes: b'\xe3' Data: 227
000000.301 TX 0000 E3
Data in Bytes: b'e' Data: 181
000000.401 TX 0000 65
Data in Bytes: b'' Data: 39
000000.502 TX 0000 27
Data in Bytes: b',' Data: 44
000000.602 TX 0000 2C
Data in Bytes: b'\x81' Data: 129
000000.702 TX 0000 81
Data in Bytes: b'\x07' Data: 7
000000.803 TX 0000 07
Data in Bytes: b'\x8a' Data: 138
000000.903 TX 0000 8A
Data in Bytes: b'\xb6' Data: 182
000001.003 TX 0000 B6
Data in Bytes: b'\x11' Data: 17
000001.104 TX 0000 11
Data in Bytes: b'k' Data: 107
000001.204 TX 0000 68
Data in Bytes: b'6' Data: 54
000001.304 TX 0000 36
Data in Bytes: b'\x18' Data: 24
000001.404 TX 0000 18
Data in Bytes: b'1' Data: 49
000001.505 TX 0000 31
Data in Bytes: b'\xd0' Data: 208
000001.605 TX 0000 D0
Data in Bytes: b'\xf6' Data: 246
000001.705 TX 0000 F6
Data in Bytes: b'\xa5' Data: 165
000001.806 TX 0000 A5
Data in Bytes: b'\xd3' Data: 211
000001.906 TX 0000 D3
Data in Bytes: b'\xc8' Data: 200
000002.006 TX 0000 C8
Data in Bytes: b'X' Data: 88
000002.107 TX 0000 58
Data in Bytes: b'^' Data: 126
000002.207 TX 0000 7E
Data in Bytes: b'\x94' Data: 148
000002.307 TX 0000 94
Data in Bytes: b'k' Data: 107
000002.407 TX 0000 68
Data in Bytes: b'S' Data: 83
000002.508 TX 0000 53
Data in Bytes: b'\x0b' Data: 11
000002.608 TX 0000 0B
Data in Bytes: b'y' Data: 121
000002.708 TX 0000 79
Data in Bytes: b'W' Data: 87
000002.809 TX 0000 57
Data in Bytes: b'T' Data: 84
000002.909 TX 0000 54
Data in Bytes: b'1' Data: 49
000003.009 TX 0000 31
Data in Bytes: b'\x07' Data: 7
000003.110 TX 0000 07
Data in Bytes: b'\xf1' Data: 241
000003.210 TX 0000 F1
Data in Bytes: b'^' Data: 94
000003.310 TX 0000 5E
Data in Bytes: b'X' Data: 88
000003.411 TX 0000 58
000004.512 RX 0000 20 53 63 68 6F 65 6E 65 20 43 72 79 70 74 6F
000004.512 RX 0010 29 57 65 6C 74 8D 0D 0D 0D 0D 0D 0D 0D 0D 0D
000004.512 RX 0020 0D 0D 0D 0D
Replb'D Schoene Crypto Welt{r|r|r|r|r|r|r|r|r|r|r|r|r|r|x00'
ok

```

PlunserFabio

```
test_05 (_main__MyTests) ... 000000.000 TX      0000  4B
000000.100 TX      0000  41          A
000000.201 TX      0000  41          A
000000.301 TX      0000  41          A
000000.401 TX      0000  41          A
000000.501 TX      0000  41          A
000000.602 TX      0000  41          A
000000.702 TX      0000  41          A
000000.802 TX      0000  41          A
000000.903 TX      0000  41          A
000001.003 TX      0000  41          A
000001.103 TX      0000  41          A
000001.203 TX      0000  41          A
000001.304 TX      0000  41          A
000001.404 TX      0000  41          A
000001.504 TX      0000  41          A
000001.605 TX      0000  41          A
000001.705 TX      0000  58          X
000001.905 TX      0000  49          I
000002.006 TX      0000  41          A
000002.106 TX      0000  41          A
000002.206 TX      0000  41          A
000002.306 TX      0000  41          A
000002.407 TX      0000  41          A
000002.507 TX      0000  41          A
000002.607 TX      0000  41          A
000002.707 TX      0000  41          A
000002.808 TX      0000  41          A
000002.908 TX      0000  41          A
000003.008 TX      0000  41          A
000003.109 TX      0000  41          A
000003.209 TX      0000  41          A
000003.309 TX      0000  41          A
000003.409 TX      0000  41          A
000003.510 TX      0000  41          A
000003.610 TX      0000  58          X
000003.710 TX      0000  44          D
000003.811 TX      0000  20          .
000003.911 TX      0000  55          U
000004.011 TX      0000  8F          .
000004.111 TX      0000  85          .
000004.212 TX      0000  68          h
000004.312 TX      0000  96          .
000004.412 TX      0000  87          .
000004.513 TX      0000  31          l
000004.613 TX      0000  42          B
000004.713 TX      0000  B1          .
000004.813 TX      0000  6D          m
000004.914 TX      0000  C8          .
000005.014 TX      0000  F2          .
000005.114 TX      0000  EA          .
000005.215 TX      0000  8A          .
000005.315 TX      0000  33          3
000005.415 TX      0000  4E          N
000005.515 TX      0000  DA          .
000005.616 TX      0000  7E          ~
000005.716 TX      0000  8F          .
000005.816 TX      0000  71          q
000005.917 TX      0000  37          7
000006.017 TX      0000  87          .
000006.117 TX      0000  7E          ~
000006.218 TX      0000  C2          .
000006.318 TX      0000  50          P
000006.418 TX      0000  AD          .
000006.518 TX      0000  73          s
000006.619 TX      0000  3A          :
000006.719 TX      0000  74          t
000006.819 TX      0000  03          .
000006.920 TX      0000  CF          .
000007.020 TX      0000  C0          .
000007.120 TX      0000  58          X
000008.222 RX      0000  44 20 53 63 68 6F 65 6E   65 28 43 72 79 70 74 6F  D Schoene Crypto
000008.222 RX      0010  20 57 65 6C 74 80 8D 8D   8D 8D 8D 8D 8D 8D 8D 8D  Welt.....
000008.222 RX      0020  8D 8D 80             ....
Reply'b'D Schoene Crypto Welt'r'r'r'r'r'r'r'r'r'r\r\x00'
ok
```

Page 18 of 18