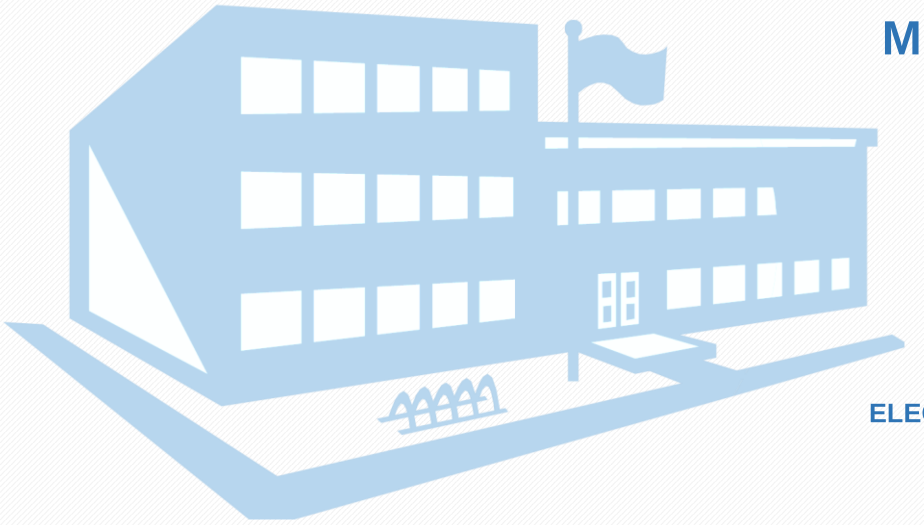
ADVANCED MICROPROCESSORS



Project Εξαμήνου

UNIVERSITY OF PATRAS
ENGINEERING SCHOOL
ELECTRICAL COMPUTER ENGINEERING
DEPARTMENT

Secure communication between 2 ARM boards implementing AES-128 in ARM THUMB assembly for low power.

Name:	ID:	Dept:	Year:
Nikolaos Tsafas	8219	ECE	5th
Nikolaos Skamnelos	1041878	CEID	3rd

Fall Semester 2016-2017

Table of Contents

ABSTRACT

This project describes a secure system of communication between two computers and two arm boards implementing AES 128-bit CBC cryptography for the Cortex-M4 embedded microprocessors exclusively in Assembly ARM THUMB-2 code.

1. INTRODUCTION

Security is one of the most important factors in our life. We apply password to our PC's, laptops for protecting our private data. Information is an asset and an asset needs to be secured from attacks. To be secured, information needs to be hidden from unauthorized access, protected from unauthorized change and availability to an authorized entity when needed.

Data security is very important especially from secured transmission of data point of view. Cryptography is a one of the various techniques for transmitting data securely.

For these reasons we present a secure communication system for the most popular modern microprocessor for embedded devices, the ARM Cortex-M4.

Sometimes an embedded device contains a coprocessor that can perform AES encryption in hardware, but such a coprocessor is not always available especially in low power implementations such as Cortex-M0 or even versions of Cortex-M4. Simply compiling an existing implementation written in, for example, the C programming language, is unlikely to produce optimal performance. For this reason, we wrote our implementation exclusively in Assembly ARM THUMB-2 code with optimization regarding loop unfolding and usage of exclusively thumb 16-bit instructions for code density and speed resulting in a suitable for low area and low resources scenario.

2. DESIGN OVERVIEW

2.1 ARM CORTEX-M

The Cortex-M is a family of 32-bit processors by ARM meant for use in embedded microcontrollers. They are designed to be cheap and to be energy efficient, while still being powerful enough to offer adequate performance in applications such as automotive systems, medical instruments, the Internet of Things, or other consumer products.

We used the STM32F407VG development board which has a 168MHz Cortex-M4 core, 1024KB of flash memory and 192KB of RAM.
The STM32F407 embeds four universal synchronous/asynchronous receiver transmitters and two universal asynchronous receiver transmitters, as well as, a USB OTG full-speed device/host/OTG peripheral with integrated transceivers. The USB OTG FS peripheral is compliant with the USB 2.0 specification and with the OTG 1.0 specification. It has software-configurable endpoint setting and supports suspend/resume. The USB OTG full-speed controller requires a dedicated 48 MHz clock that is generated by a PLL connected to the HSE oscillator.

On the STM32F407VG, there are many regions to store data: SRAM, FLASH and CCM. In our case we used the SRAM to increase performance.

2.2 AES

The Advanced Encryption Standard (AES), also known as Rijndael, is a specification for the encryption of electronic data established by the U.S. National Institute of Standards and Technology (NIST) in 2001. AES has been adopted by the U.S. government and is now used worldwide. It supersedes the Data Encryption Standard (DES), which was published in 1977.

The algorithm described by AES is a symmetric-key algorithm, meaning the same key is used for both encrypting and decrypting the data.

2.2.1 Algorithm

AES is based on a design principle known as a substitution-permutation network, combination of both substitution and permutation, and is fast in both software and hardware.

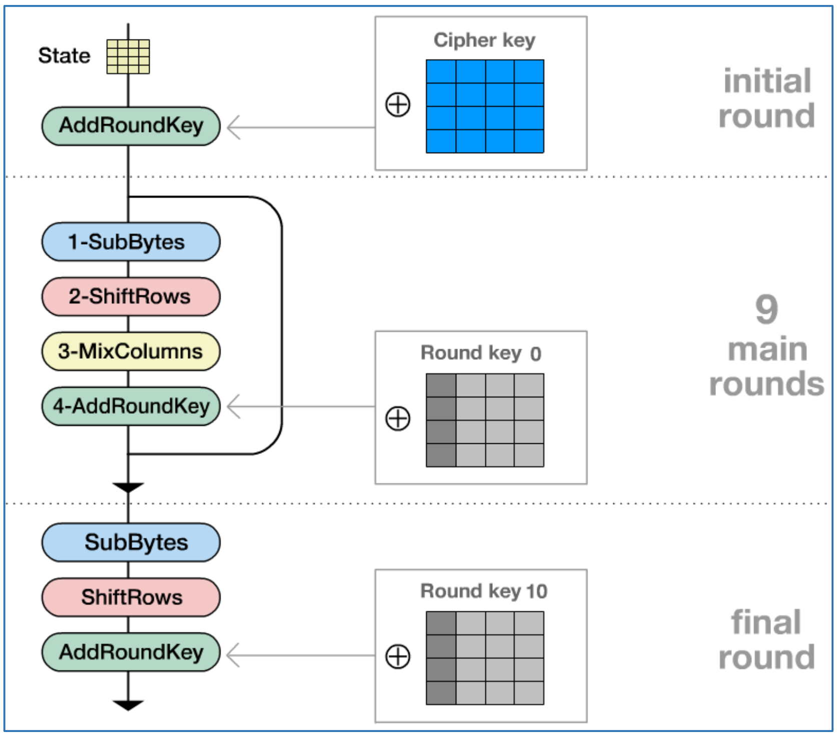
The cipher has a fixed block size of 128 bits, and operates on a 4 × 4 column-major order matrix of bytes, termed "**the state**".

The key size used for an AES cipher specifies the number of repetitions of transformation rounds that convert the input, called the plaintext, into the final output, called the ciphertext. The number of cycles of repetition are as follows:

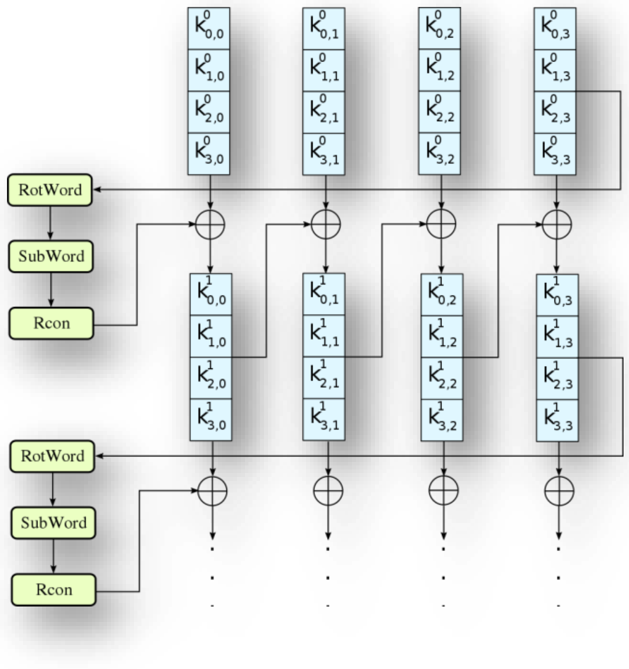
- 10 cycles of repetition for 128-bit keys.
- 12 cycles of repetition for 192-bit keys.
- 14 cycles of repetition for 256-bit keys.

One that depends on the encryption key itself and a set of rounds each containing four similar but different stages

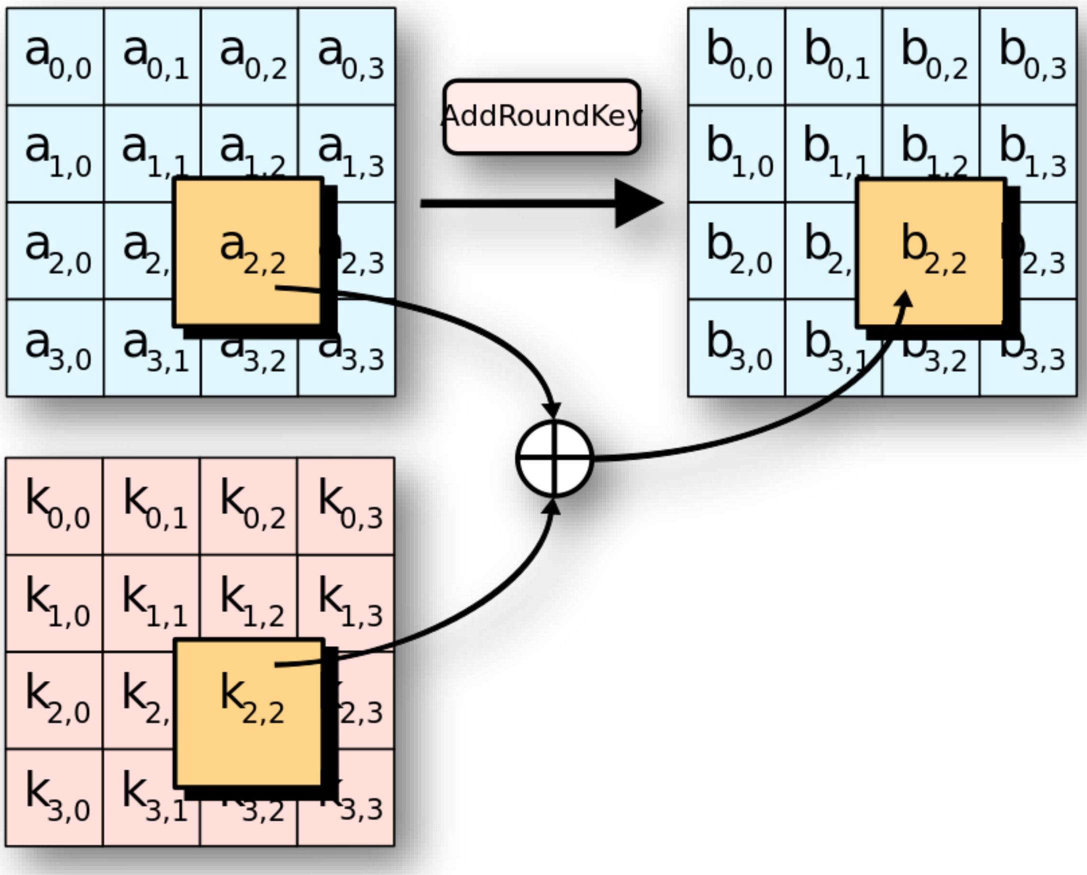
Each round consists of several processing steps, each containing four similar but different stages



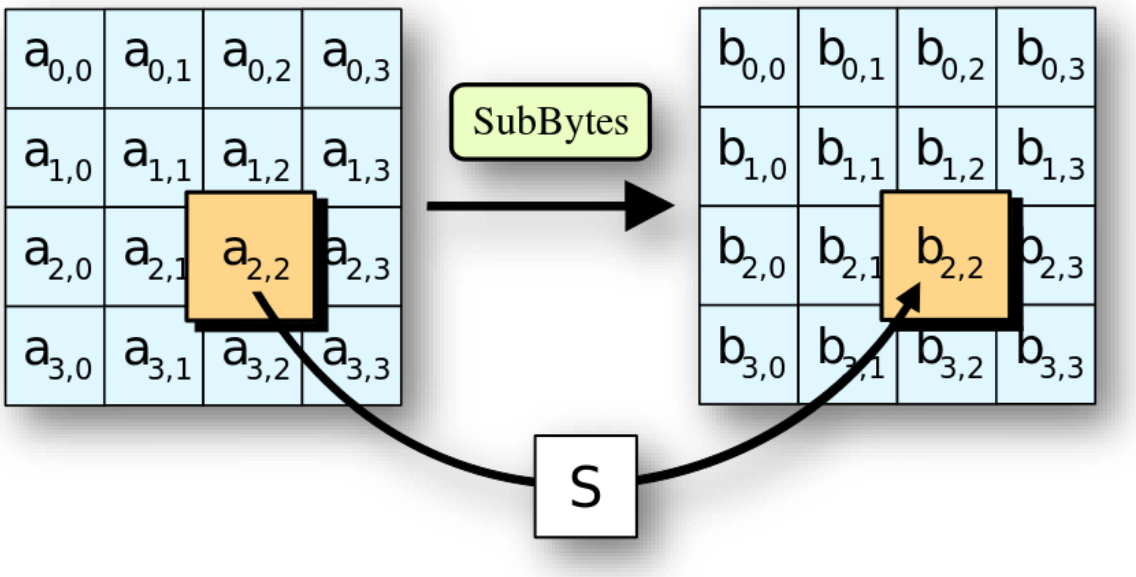
a) Key Expansion:
Round keys are derived from the cipher key using Rijndael's key schedule. AES requires a separate 128-bit round key block for each round plus one more.



b) Add Round Key:
Each byte of the state is combined with a block of the round key using bitwise xor.

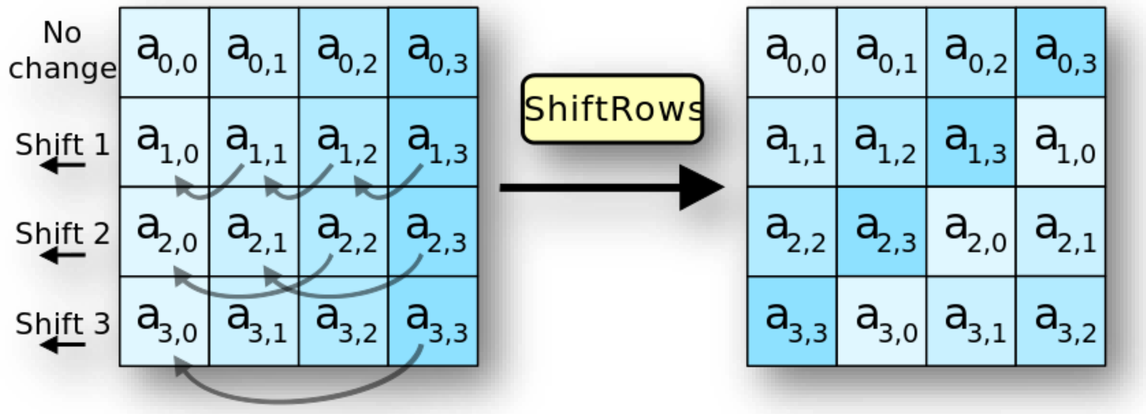


c) Sub Bytes:
In the SubBytes step, each byte in the state matrix is replaced with a SubByte using an 8-bit substitution box, the Rijndael S-box. This operation provides the non-linearity in the cipher.



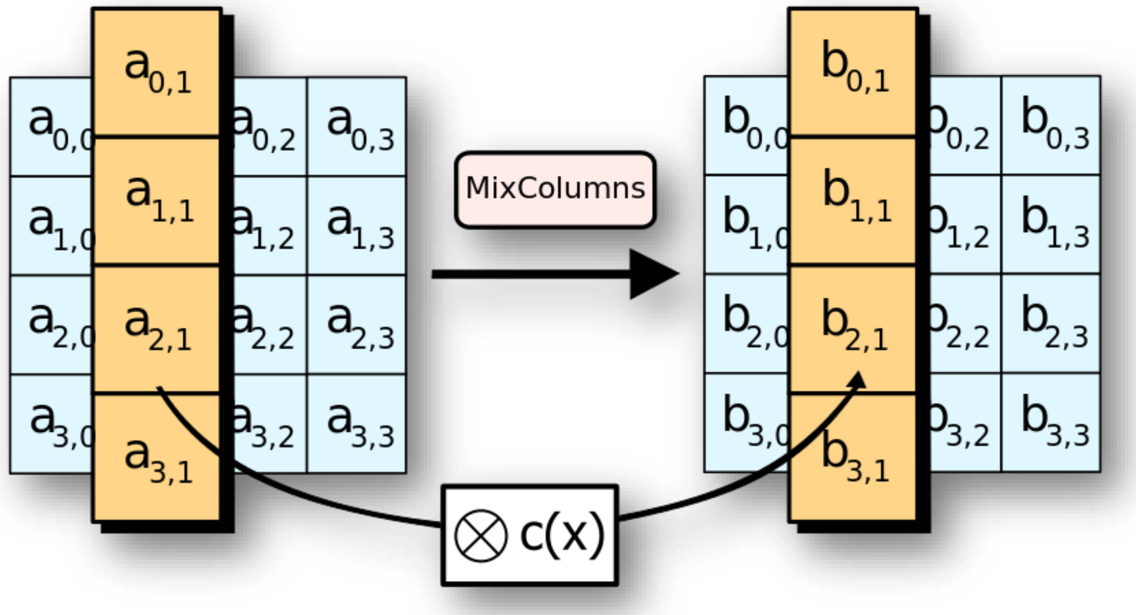
$$\begin{array}{cccc} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{array} \xrightarrow{\text{SubBytes}} \begin{array}{cccc} b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} \\ b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} \end{array}$$

d) Shift Row:
A transposition step where the last three rows of the state are shifted cyclically a certain number of steps.



$$\begin{array}{r|cccc} \text{No change} & a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ \text{Shift 1} & a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ \text{Shift 2} & a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ \text{Shift 3} & a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{array} \xrightarrow{\text{ShiftRows}} \begin{array}{cccc} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,1} & a_{1,2} & a_{1,3} & a_{1,0} \\ a_{2,2} & a_{2,3} & a_{2,0} & a_{2,1} \\ a_{3,3} & a_{3,0} & a_{3,1} & a_{3,2} \end{array}$$

e) Mix Column:
In the Mix Columns step, each column of the state is multiplied with a fixed polynomial c (x) .



$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$
$a_{1,0}$ $a_{1,1}$ $a_{1,2}$ $a_{1,3}$
$a_{2,0}$ $a_{2,1}$ $a_{2,2}$ $a_{2,3}$
$a_{3,0}$ $a_{3,1}$ $a_{3,2}$ $a_{3,3}$
MixColumns
$\otimes$ c(x)
$b_{0,0}$ $b_{0,1}$ $b_{0,2}$ $b_{0,3}$
$b_{1,0}$ $b_{1,1}$ $b_{1,2}$ $b_{1,3}$
$b_{2,0}$ $b_{2,1}$ $b_{2,2}$ $b_{2,3}$
$b_{3,0}$ $b_{3,1}$ $b_{3,2}$ $b_{3,3}$

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

This can also be seen as the following:

$$\begin{aligned} b_0 &= 2a_0 + 3a_1 + 1a_2 + 1a_3 \\ b_1 &= 1a_0 + 2a_1 + 3a_2 + 1a_3 \\ b_2 &= 1a_0 + 1a_1 + 2a_2 + 3a_3 \\ b_3 &= 3a_0 + 1a_1 + 1a_2 + 2a_3 \end{aligned}$$

Respectively the decryption process is described by the following schematic, in which the processes within each round are executed in reverse order and each function has been changed by an inverted one:

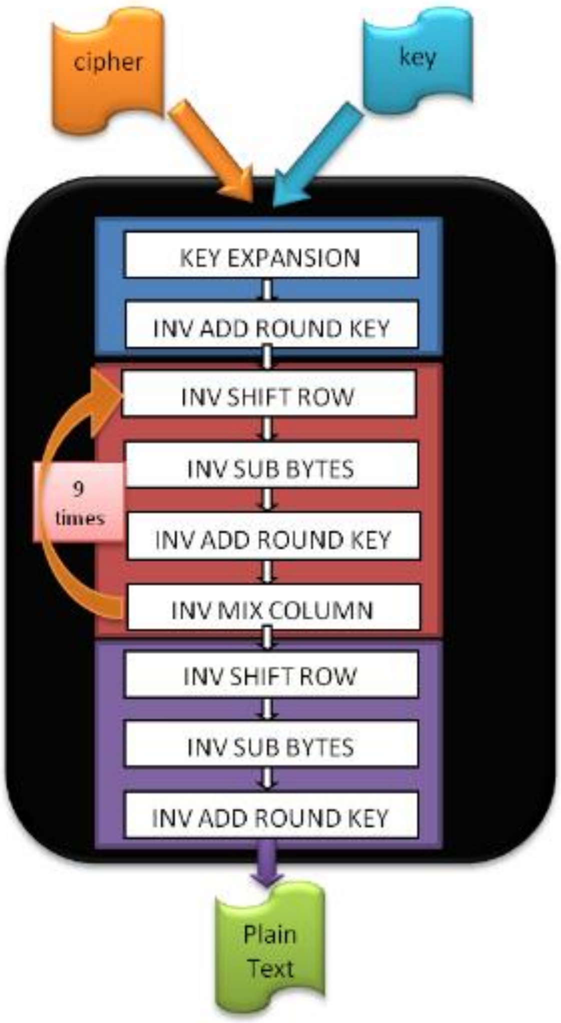


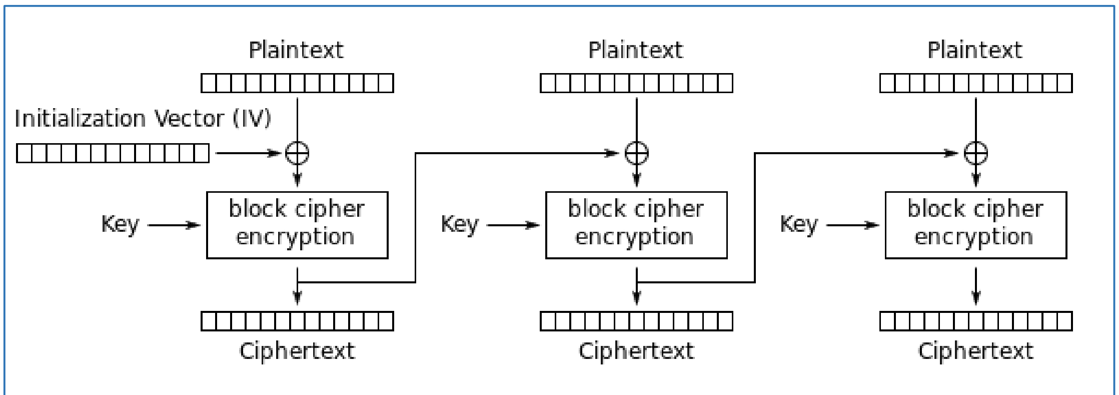
Figure 3 - AES 128-bit Decryption Algorithm

2.2.2 Mode of operation

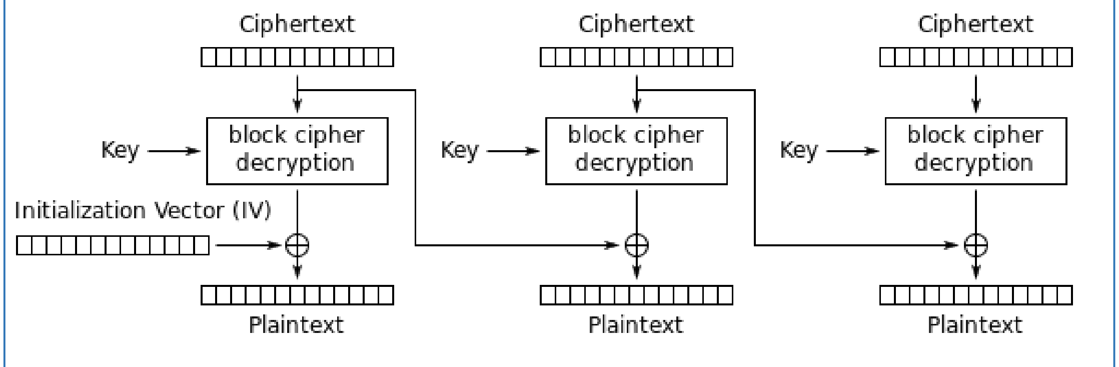
A mode of operation describes how to repeatedly apply a cipher's single-block operation to securely transform amounts of data larger than a block.

Cipher Block Chaining (CBC):

In CBC mode, each block of plaintext is XORed with the previous ciphertext block before being encrypted. This way, each ciphertext block depends on all plaintext blocks processed up to that point. To make each message unique, an initialization vector must be used in the first block.



Cipher Block Chaining (CBC) mode encryption



Cipher Block Chaining (CBC) mode decryption

2.2.3 Padding

Block ciphers have one or more block size(s), but during transformation the block size is always fixed. Block cipher modes operate on whole blocks and require that the last part of the data be padded to a full block if it is smaller than the current block size.

There are at least five common conventions:

1) Pad with bytes all of the same value as the number of padding bytes
2) Pad with 0x80 followed by zero bytes
3) Pad with zeroes except make the last byte equal to the number of padding bytes
4) Pad with zero (null) characters
5) Pad with space characters

Method one is the most popular and is usually referred to as "PKCS5 padding". It is generally recommended in the absence of any other considerations.

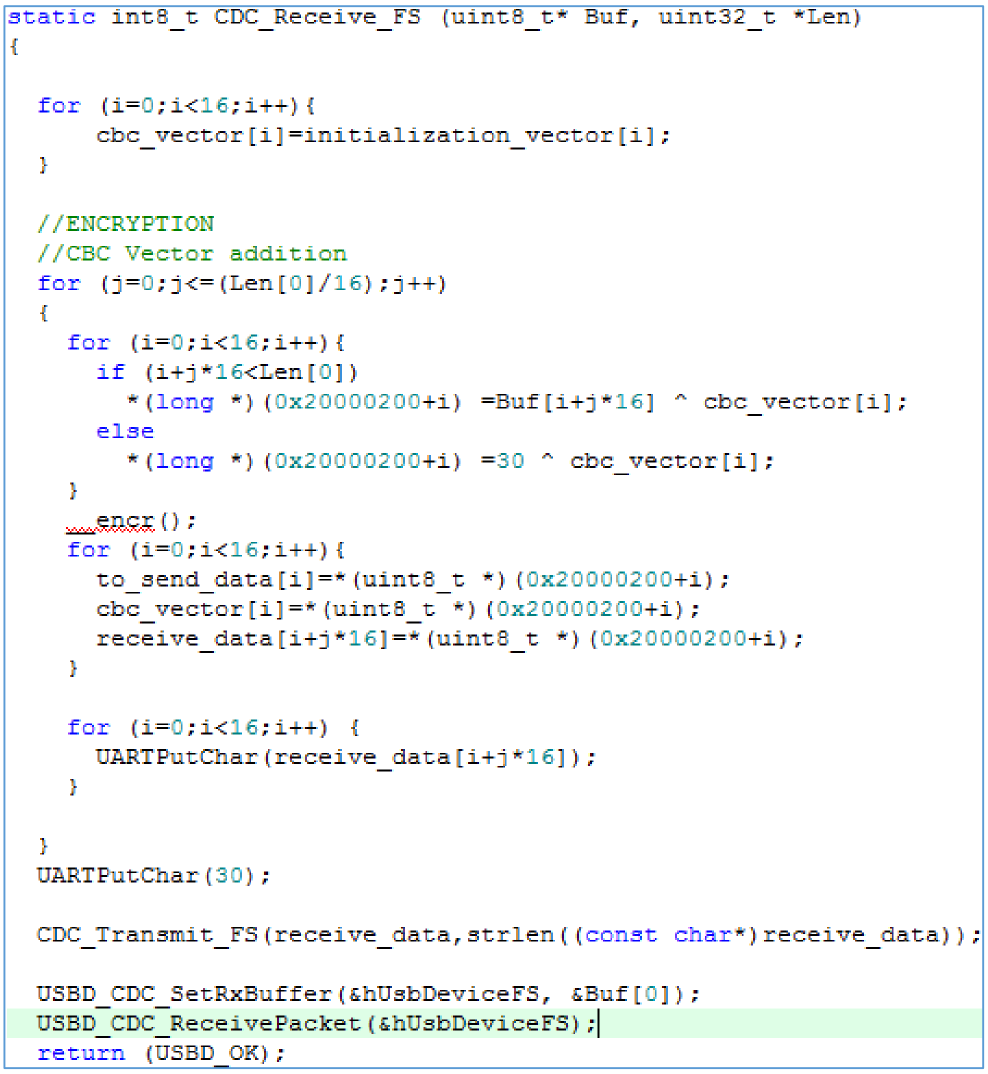
2.3 USB COMMUNICATION

USB, short for Universal Serial Bus, is an industry standard initially developed in the mid-1990s that defines the cables, connectors and communications protocols used in a bus for connection, communication, and power supply between computers and electronic devices.

The functionality of USB devices is defined by class codes. The communications device class is used for computer networking devices. Devices of this class are also implemented in embedded systems. The data interfaces are generally used to perform bulk data transfer.

In order to implement this module in our project we used the HAL libraries provided by ST.

Below our usb receive and transmit function implementation is shown.



```
static int8_t CDC_Receive_FS (uint8_t* Buf, uint32_t *Len)
{

  for (i=0;i<16;i++){
      cbc_vector[i]=initialization_vector[i];
  }

  //ENCRYPTION
  //CBC Vector addition
  for (j=0;j<=(Len[0]/16);j++)
  {
    for (i=0;i<16;i++){
      if (i+j*16<Len[0])
        *(long *)(0x20000200+i) =Buf[i+j*16] ^ cbc_vector[i];
      else
        *(long *)(0x20000200+i) =30 ^ cbc_vector[i];
    }
      encr();
    for (i=0;i<16;i++){
      to_send_data[i]=*(uint8_t *)(0x20000200+i);
      cbc_vector[i]=*(uint8_t *)(0x20000200+i);
      receive_data[i+j*16]=*(uint8_t *)(0x20000200+i);
    }

    for (i=0;i<16;i++) {
      UARTPutChar(receive_data[i+j*16]);
    }

  }
  UARTPutChar(30);

  CDC_Transmit_FS(receive_data,strlen((const char*)receive_data));

  USBD_CDC_SetRxBuffer(&hUsbDeviceFS, &Buf[0]);
  USBD_CDC_ReceivePacket(&hUsbDeviceFS);
  return (USBD_OK);
```

```
uint8_t CDC_Transmit_FS(uint8_t* Buf, uint16_t Len)
{
  uint8_t result = USBD_OK;

  memcpy(UserTxBufferFS,Buf,sizeof(uint8_t)*Len);
  USBD_CDC_HandleTypeDef *hcdc = (USBD_CDC_HandleTypeDef*)hUsbDeviceFS.pClassData;
  if (hcdc->TxState != 0){
    return USBD_BUSY;
  }
  USBD_CDC_SetTxBuffer(&hUsbDeviceFS, Buf, Len);
  result = USBD_CDC_TransmitPacket(&hUsbDeviceFS);
  /* USER CODE END 7 */
  return result;
```

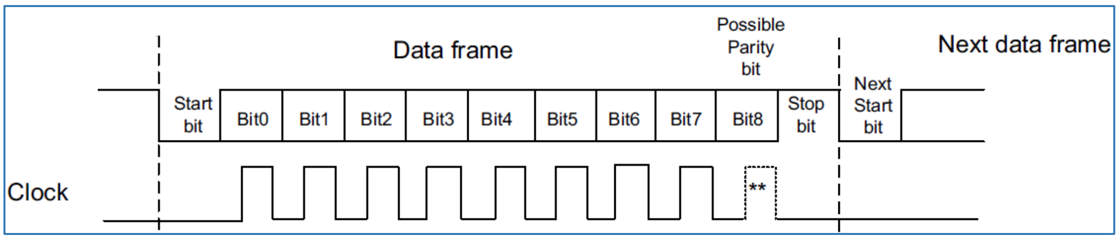
2.4 UART COMMUNICATION

2.4.1 Description

The universal synchronous asynchronous receiver transmitter (USART) offers a flexible means of full-duplex data exchange with external equipment using an asynchronous serial data format.

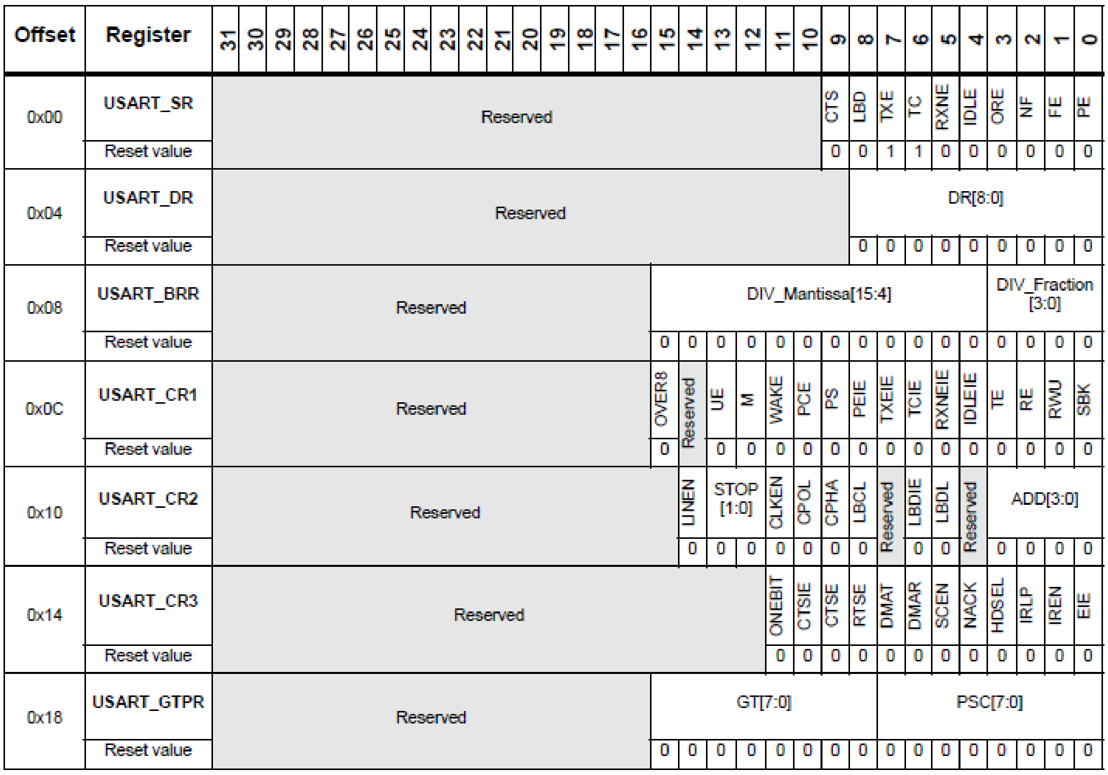
USART bidirectional communication requires a minimum of two pins: Receive Data In (RX) and Transmit Data Out (TX).

The transmitter can send data words of either 8 or 9 bits. Every character is preceded by a start bit which is a logic level low for one-bit period. The character is terminated by a configurable number of stop bits. The following stop bits are supported by USART: 0.5, 1, 1.5 and 2 stop bits.



2.4.2 Configuration

This project utilizes USART6 of the board (USART6_RX : pin PC7 USART6_RX : pin PC6) located in address “0x40011400”. The registers of the USART are shown below.



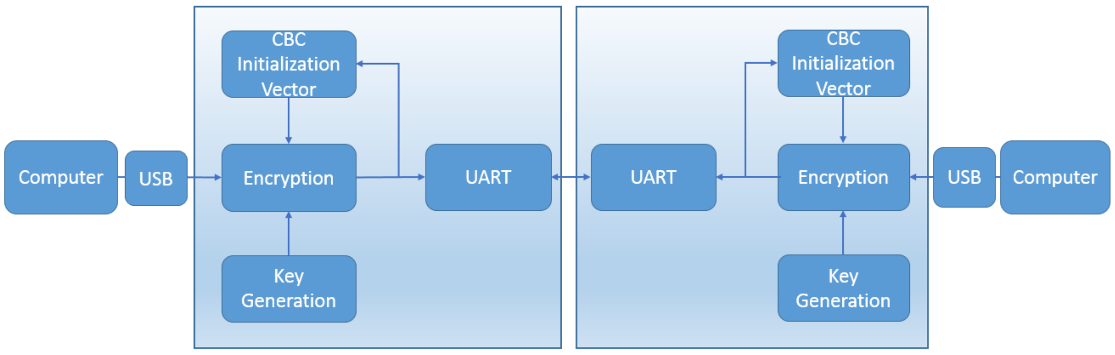
Offset	Register	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x00	USART_SR	Reserved																						CTS	LBD	TXE	TC	RXNE	IDLE	ORE	NF	FE	PE
	Reset value																							0	0	1	1	0	0	0	0	0	0
0x04	USART_DR	Reserved																							DR[8:0]								
	Reset value																								0	0	0	0	0	0	0	0	0
0x08	USART_BRR	Reserved																DIV_Mantissa[15:4]												DIV_Fraction [3:0]			
	Reset value																	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x0C	USART_CR1	Reserved																OVER8	Reserved	UE	M	WAKE	PCE	PS	PEIE	TXEIE	TCIE	RXNEIE	IDLEIE	TE	RE	RWU	SBK
	Reset value																	0		0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x10	USART_CR2	Reserved																	LINEN	STOP [1:0]		CLKEN	CPOL	CPHA	LBCL	Reserved	LBDIE	LBDL	Reserved	ADD[3:0]			
	Reset value																		0	0	0	0	0	0	0		0	0		0	0	0	0
0x14	USART_CR3	Reserved																				ONEBIT	CTSIE	CTSE	RTSE	DMAT	DMAR	SCEN	NACK	HDSEL	IRLP	IREN	EIE
	Reset value																					0	0	0	0	0	0	0	0	0	0	0	0
0x18	USART_GTPR	Reserved																GT[7:0]								PSC[7:0]							
	Reset value																	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

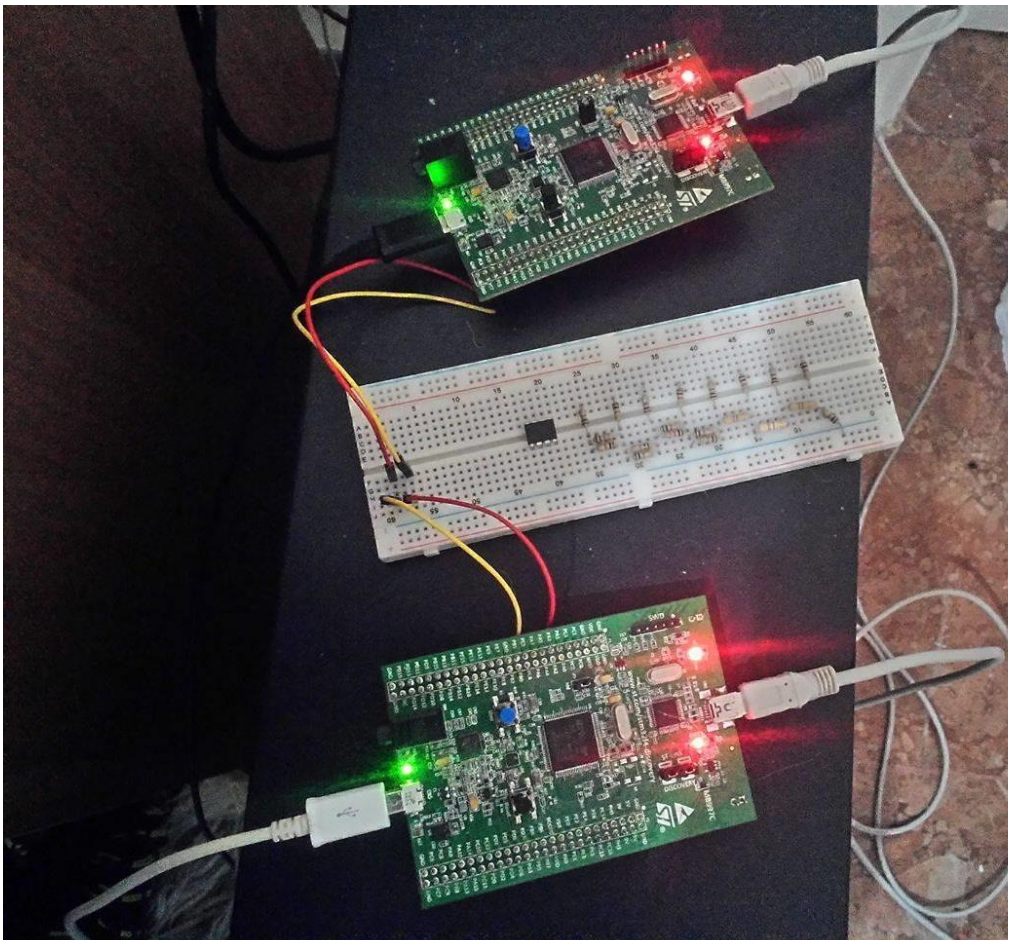
Procedure:
1) Enable the USART by writing the UE bit in USART_CR1 register to 1.
2) Program the M bit in USART_CR1 to define the word length.
3) Program the number of stop bits in USART_CR2.
4) Select DMA enable (DMAT) in USART_CR3 if Multi buffer Communication is to take place. Configure the DMA register as explained in multibuffer communication.
5) Select the desired baud rate using the USART_BRR register.
6) Set the TE bit in USART_CR1 to send an idle frame as first transmission.
7) Write the data to send in the USART_DR register (this clears the TXE bit). Repeat this for each data to be transmitted in case of single buffer.
8) After writing the last data into the USART_DR register, wait until TC=1. This indicates that the transmission of the last frame is complete. This is required for instance when the USART is disabled or enters the Halt mode to avoid corrupting the last transmission.

```
void ConfigureUART(unsigned int baudDivisor)
  {USART6_CR1 = 0;            //Disable during set up. Wd len = 8, Parity = off
   USART6_BRR = baudDivisor;  //Set up baud rate
   USART6_CR2 = 0x2000;            //1 stop bit
   USART6_CR1 = 0x2004;
   USART6_CR3 = 0;            //Disable interrupts and DMA
  }

void UARTPutChar(char ch)
  {//Wait for empty flag
   while((USART6_SR & 0x80) == 0);
   USART6_DR = ch;
  }

char UARTReadChar()
{//Wait for empty flag
  while((USART6_SR & 0x20) != 0x20);
  return USART6_DR;
}
```

3. IMPLEMENTATION

Our implementation is presented in the following schematic.





3.1 Description

A user sends a message through usb to the application board.

The encryption process begins with the shared key and a standard Initialization Vector and using CBC mode it sends each block through UART.

The second board performs the decryption and sends the decrypted message to the second user.

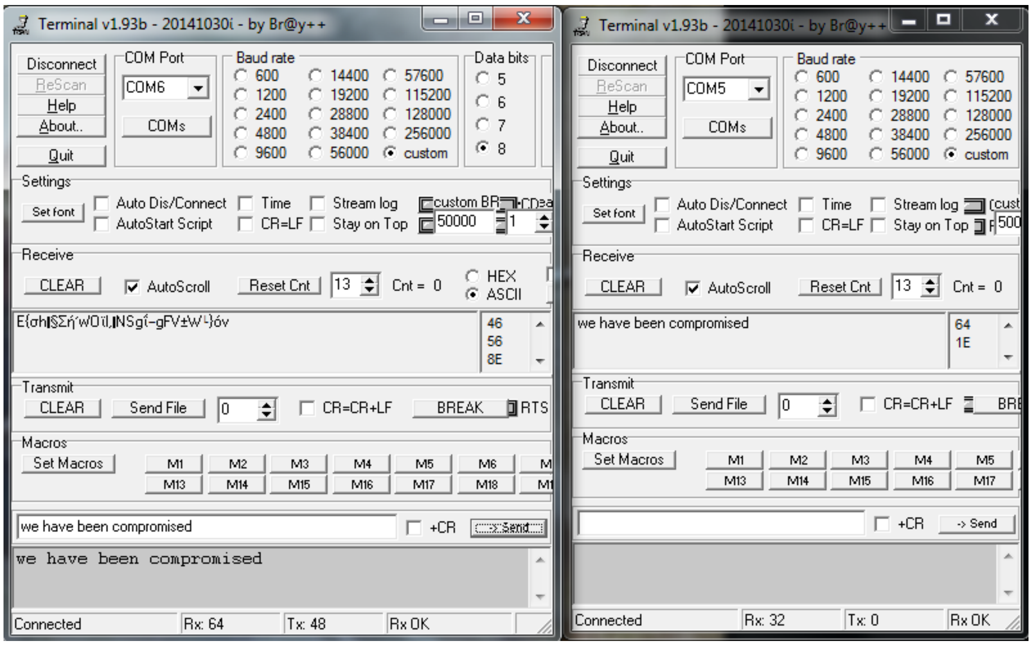
This scheme can work in either direction as both boards are equipped with encryption-decryption and transmit-receive functions.

3.2 Operation

We send the message "we have been compromised"

In the left window we see the ciphertext in asci while in the right window we see the decrypted message in the receiver end.

In this example the cbc mode of operation is checked if working properly as can be also seen in debugging mode of the processor in the next image.



Terminal v1.93b - 20141030ί - by Br@y++

COM Port: COM6
Baud rate: custom
Data bits: 8
custom BR: 50000

Receive: AutoScroll ✓, ASCII, Cnt = 0

```
E{σh|§Σή´wOïl,|NSgί–gFV±W└}όv
```

46 56 8E

Transmit:

```
we have been compromised
```

```
we have been compromised
```

Connected Rx: 64 Tx: 48 Rx OK

Terminal v1.93b - 20141030ί - by Br@y++

COM Port: COM5
Baud rate: custom

Receive: AutoScroll ✓, Cnt = 0

```
we have been compromised
```

64 1E

Connected Rx: 32 Tx: 0 Rx OK

In the following images the encryption process is presented in the memory of the arm board.

We put the first block in 0x20000200

While in 0x20000220 we have the expanded key

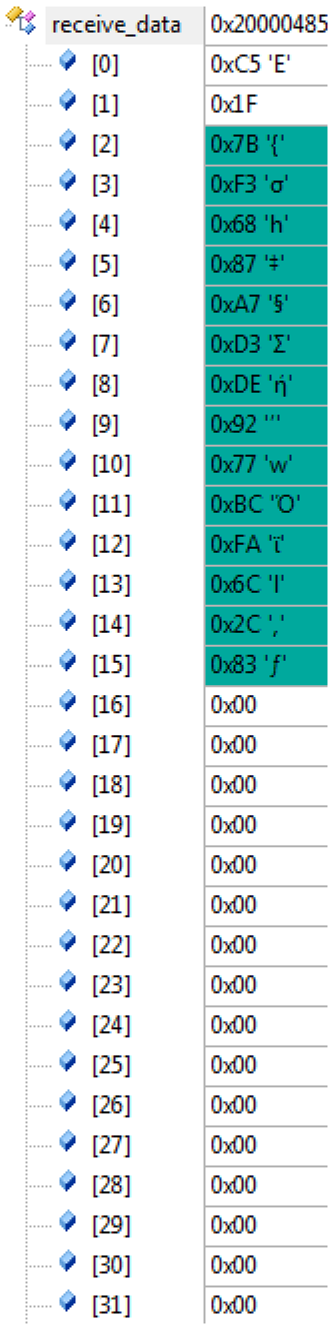
```
0x20000200: 77 74 02 5B 25 23 03 57 EA FC CF D5 EC BE 81 92
0x20000210: 00 00 00 00 00 00 D8 32 00 08 D8 33 00 08 00 00
0x20000220: 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
0x20000230: D6 AA 74 FD D2 AF 72 FA DA A6 78 F1 D6 AB 76 FE
0x20000240: B6 92 CF 0B 64 3D BD F1 BE 9B C5 00 68 30 B3 FE
0x20000250: B6 FF 74 4E D2 C2 C9 BF 6C 59 0C BF 04 69 BF 41
0x20000260: 47 F7 F7 BC 95 35 3E 03 F9 6C 32 BC FD 05 8D FD
0x20000270: 3C AA A3 E8 A9 9F 9D EB 50 F3 AF 57 AD F6 22 AA
0x20000280: 5E 39 0F 7D F7 A6 92 96 A7 55 3D C1 0A A3 1F 6B
0x20000290: 14 F9 70 1A E3 5F E2 8C 44 0A DF 4D 4E A9 C0 26
0x200002A0: 47 43 87 35 A4 1C 65 B9 E0 16 BA F4 AE BF 7A D2
0x200002B0: 54 99 32 D1 F0 85 57 68 10 93 ED 9C BE 2C 97 4E
0x200002C0: 13 11 1D 7F E3 94 4A 17 F3 07 A7 8B 4D 2B 30 C5
```

Next we see the two encrypted blocks after the 2 iterations as the message length is 2 blocks.

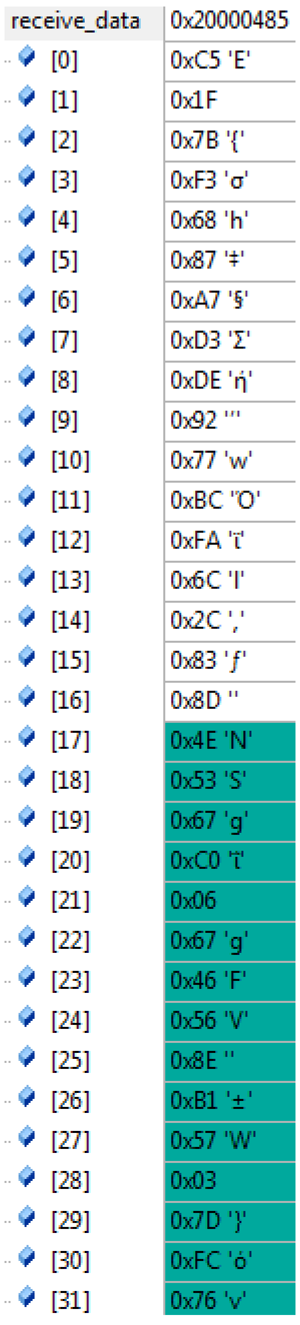
```
0x20000200: 8D 4E 53 67 C0 06 67 46 56 8E B1 57 03 7D FC 76
```

```
0x20000200: C5 1F 7B F3 68 87 A7 D3 DE 92 77 BC FA 6C 2C 83
```

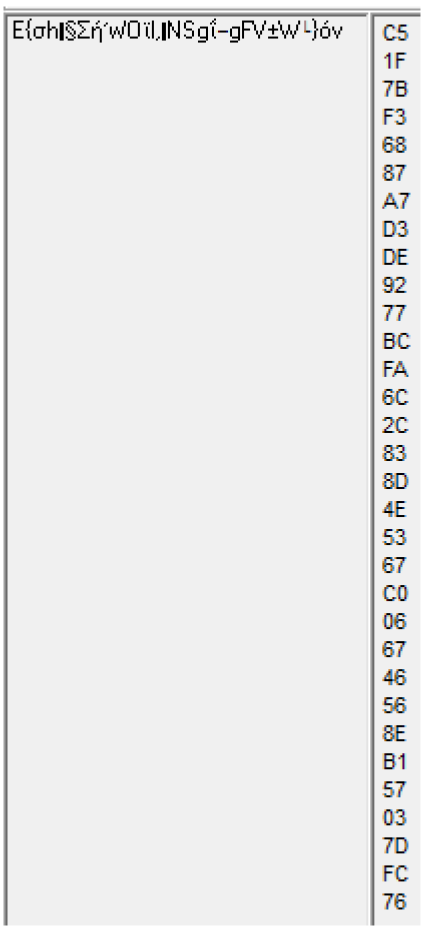
Also we see how the variables are updated after each block operation and the final ciphertext transmitted through UART.



```
receive_data   0x20000485
  [0]          0xC5 'E'
  [1]          0x1F
  [2]          0x7B '{'
  [3]          0xF3 'σ'
  [4]          0x68 'h'
  [5]          0x87 '‡'
  [6]          0xA7 '§'
  [7]          0xD3 'Σ'
  [8]          0xDE 'ή'
  [9]          0x92 '''
  [10]         0x77 'w'
  [11]         0xBC 'Ό'
  [12]         0xFA 'ϊ'
  [13]         0x6C 'l'
  [14]         0x2C ','
  [15]         0x83 'ƒ'
  [16]         0x00
  [17]         0x00
  [18]         0x00
  [19]         0x00
  [20]         0x00
  [21]         0x00
  [22]         0x00
  [23]         0x00
  [24]         0x00
  [25]         0x00
  [26]         0x00
  [27]         0x00
  [28]         0x00
  [29]         0x00
  [30]         0x00
  [31]         0x00
```



```
receive_data   0x20000485
  [0]          0xC5 'E'
  [1]          0x1F
  [2]          0x7B '{'
  [3]          0xF3 'σ'
  [4]          0x68 'h'
  [5]          0x87 '‡'
  [6]          0xA7 '§'
  [7]          0xD3 'Σ'
  [8]          0xDE 'ή'
  [9]          0x92 '''
  [10]         0x77 'w'
  [11]         0xBC 'Ό'
  [12]         0xFA 'ϊ'
  [13]         0x6C 'l'
  [14]         0x2C ','
  [15]         0x83 'ƒ'
  [16]         0x8D ''
  [17]         0x4E 'N'
  [18]         0x53 'S'
  [19]         0x67 'g'
  [20]         0xC0 'ΐ'
  [21]         0x06
  [22]         0x67 'g'
  [23]         0x46 'F'
  [24]         0x56 'V'
  [25]         0x8E ''
  [26]         0xB1 '±'
  [27]         0x57 'W'
  [28]         0x03
  [29]         0x7D '}'
  [30]         0xFC 'ό'
  [31]         0x76 'v'
```

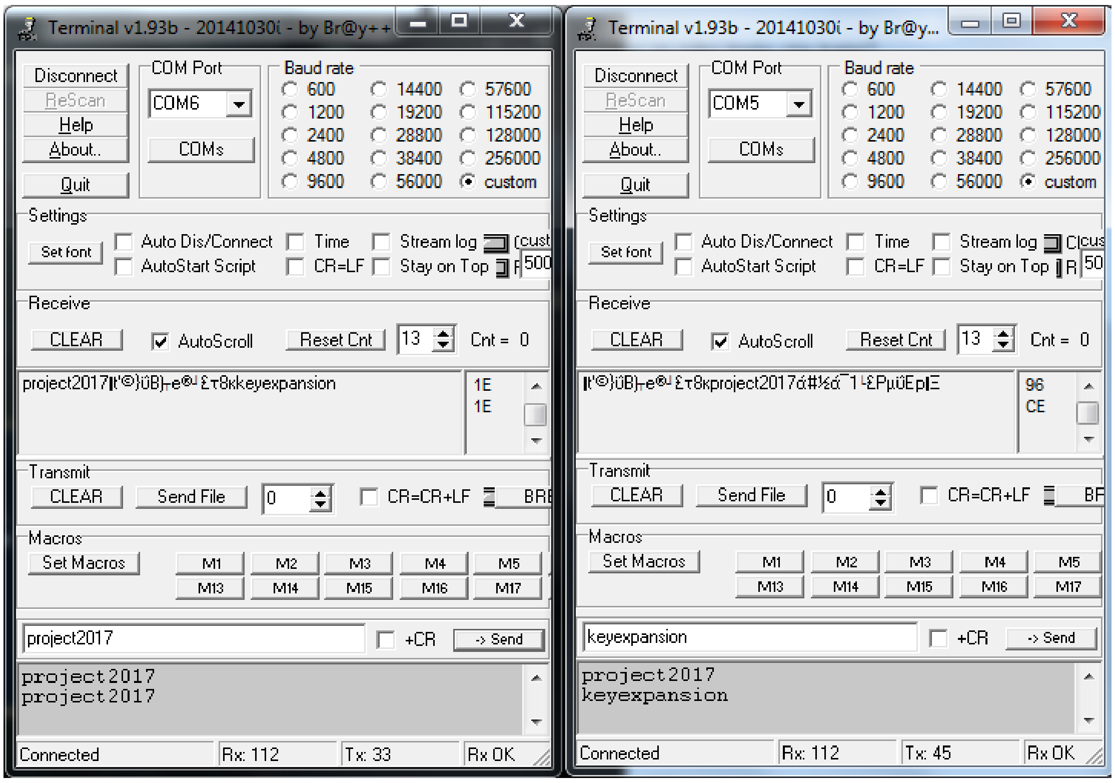


```
E{σh▌§Σή´wΌϊl,▌NSgΐ–gFV±W╘}όv      C5
                                  1F
                                  7B
                                  F3
                                  68
                                  87
                                  A7
                                  D3
                                  DE
                                  92
                                  77
                                  BC
                                  FA
                                  6C
                                  2C
                                  83
                                  8D
                                  4E
                                  53
                                  67
                                  C0
                                  06
                                  67
                                  46
                                  56
                                  8E
                                  B1
                                  57
                                  03
                                  7D
                                  FC
                                  76
```

In the next image we see an example of two way communication.

We send the message "project2017" and then the 2nd user replies "key expansion"

In both windows we see both the messages received and the ciphertext sent.



```
COM Port: COM6    Baud rate: custom    AutoScroll: ✓    13    Cnt = 0
Receive: project2017|l'©}üB}-e®ˡ£τ8κkeyexpansion    1E 1E
Transmit: project2017
project2017
project2017
Connected    Rx: 112    Tx: 33    Rx OK
```

```
COM Port: COM5    Baud rate: custom    AutoScroll: ✓    13    Cnt = 0
Receive: |l'©}üB}-e®ˡ£τ8κproject2017á#½á¯1Ŀ£PμüEp|Ξ    96 CE
Transmit: keyexpansion
project2017
keyexpansion
Connected    Rx: 112    Tx: 45    Rx OK
```

4. LESSONS LEARNED

The completition of this projects was based on partitioning of tasks and modularity enabling working simultaneously on different parts of the design, as well as, the development of skills on specification definition.

Another important aspect of our work was time management and organization with each module being implemented and tested on a finite timeline in order to produce improved working versions of our system as time progressed.

Finally, thorough study of the concepts involved in the project made us aware of the vast importance of secure communication as well as the capabilities of embedded hardware in the IoT environment.

5. EXPANDING THE IDEA

Some ideas the great team has come up with were:

- ✓ Implementation of AES 192-bit/256-bit decryption-encryption process and various other block modes of operation (Electronic Codebook (ECB), Propagating Cipher Block Chaining (PCBC), Cipher Feedback (CFB), Output Feedback (OFB), Counter (CTR)).
- ✓ Replacement of the UART communication with wireless connectivity such as, Wi-Fi or Bluetooth, for exchanging data between not only two boards but a group of users in range.
- ✓ Creation of a key distribution system that will allow the exchange of different keys between the available users.
- ✓ Using the capabilities of the ARM Microprocessor we could provide our system with a keyboard and display resulting in an autonomous secure communication device without the need of a host computer.

S

6. REFERENCES

[1] https://en.wikipedia.org/wiki/Advanced_Encryption_Standard
[2] http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf
[3] https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation#CBC
[4] https://sites.google.com/site/terminalbpp/

ABOUT THE AUTHORS

Skamnelos Nikolaos is an undergraduate student at Patras Department of Computer Engineering & Informatics. His interests lay with computer architecture and embedded software.

Tsafas Nikolaos is currently pursuing his Diploma degree in the ECE dept., University of Patras. His interests are VLSI circuit design, digital signal processing and embedded systems.