Encryption on Microcontrollers

By

Chao Chen

June 2014

A Senior Project Presented to the

Electrical Engineering Department Faculty

Of California Polytechnic State University

San Luis Obispo, California

In Partial Fulfillment

Of the Requirements for the Degree

Of Bachelors of Science

Abstract

This project concentrates on the security of simple embedded systems that sends data wirelessly. For example, a WIFI home security camera. The algorithm, Advanced Security Standard (AES), is chosen since it is a common and efficient algorithm. The processor of a simple embedded system has limited processing power and some applications are real time, so reducing the usage of the processor and time efficiency are important. The goals are achieved by minimizing processor memory usage and iteration of AES.

Table of Contents

List of Figures

Introduction

An FPGA could be used to minimize processor memory usage. However, an FPGA holds an additional cost for the embedded system. FPGA can encrypt the data faster due to parallel computing. However, encryption would be the last stage of the processing and after that data is transferred through serial communication. Thus, time is not very critical here and performing AES inside a microprocessor is a better option. Secure communication with sensitive information is necessary for military and government institutions but also for business sectors and private individuals [1]. The Rijndael algorithm was chosen by National Institute of Standards and Technology (NIST) for the new Advanced Encryption Standard in conjunction with scalability, security, simplicity, and strength. It is an iterative, symmetric block cipher operating on 128-bit block sizes with key sizes of 128, 196, and 256 bits.

Background

The 21st century relies heavily on networked infrastructure. Steps must be taken to protect data from intruders. Data encryption has the ability to defend unauthorized use of data. However, cryptology has weaknesses such as security vulnerabilities such as sending encryption keys over networks. Another weakness was found in the Data Encryption Standard (DES) 56-bit key initially adopted in 1977. The Advanced Encryption Standard (AES) was designed in response to the weak and slow DES algorithm [2]. The design methodology was for resilience against known attacks (exhaustive key search, etc) and encryption speed on CPUs, while staying as simplistic as possible [3].

Design Module

Hardware	Software
Laptop	**Advanced port terminal**
Atmega328 microcontroller	**C-code to implement the algorithm**
USB cable	**C-code to display data on the screen**

Software flow diagram:

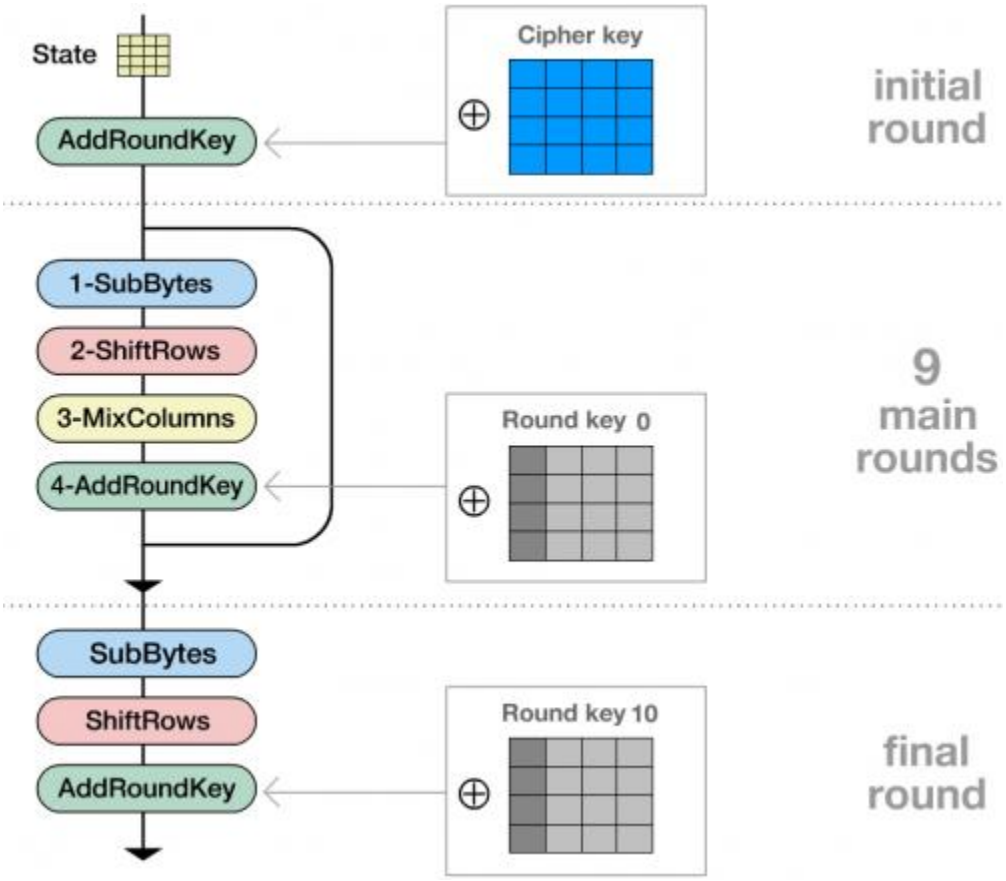

Figure 1: Traditional AES 128-bit Encryption Scheme

Traditional AES uses a repetitive encryption process to diffuse the data as shown in Figure 1. The input data will be passed through four types of stages: SubBytes, ShiftRows, MixColumns, and AddRoundKey [4].

Sub Bytes

The first stage, called Sub Bytes, is a non-linear byte substitution that acts on every byte of the state in isolation to produce a new byte value using anSbox substitution table shown in Figure 1. The corresponding replacement byte is found by matching the first nibble of an input byte to a look-up table row, and the second nibble for the look-up table column. This step helps avoid attacks based on algebraic properties.

State is represented as in Figure 2:

$S_{0,0}$	$S_{0,1}$	$S_{0,2}$	$S_{0,3}$
$S_{1,0}$	$S_{1,1}$	$S_{1,2}$	$S_{1,3}$
$S_{2,0}$	$S_{2,1}$	$S_{2,2}$	$S_{2,3}$
$S_{3,0}$	$S_{3,1}$	$S_{3,2}$	$S_{3,3}$

Figure 2: Example of the state

Figure 3 shows the Sbox of the. The first 4 bits in the byte (the first hexadecimal value, hence) individual the row, the last 4 bits individuate the column)

		y														
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
	1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
	2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
	3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
	4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
	5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
	6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
	7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
	8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
	9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
	a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
	b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
	c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
	d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
	e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
	f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Figure 3: Sbox Values for all 256 Combinations in Hexadecimal Format

An array with all 256 hexadecimal values in C code:

uint8_t s[256] = {

0x63, 0x7C, 0x77, 0x7B, 0xF2, 0x6B, 0x6F, 0xC5, 0x30, 0x01, 0x67, 0x2B, 0xFE, 0xD7, 0xAB, 0x76, 0xCA, 0x82, 0xC9, 0x7D, 0xFA, 0x59, 0x47, 0xF0, 0xAD, 0xD4, 0xA2, 0xAF, 0x9C, 0xA4, 0x72, 0xC0, 0xB7, 0xFD, 0x93, 0x26, 0x36, 0x3F, 0xF7, 0xCC, 0x34, 0xA5, 0xE5, 0xF1, 0x71, 0xD8, 0x31, 0x15, 0x04, 0xC7, 0x23, 0xC3, 0x18, 0x96, 0x05, 0x9A, 0x07, 0x12, 0x80, 0xE2, 0xEB, 0x27, 0xB2, 0x75, 0x09, 0x83, 0x2C, 0x1A, 0x1B, 0x6E, 0x5A, 0xA0, 0x52, 0x3B, 0xD6, 0xB3, 0x29, 0xE3, 0x2F, 0x84, 0x53, 0xD1, 0x00, 0xED, 0x20, 0xFC, 0xB1, 0x5B, 0x6A, 0xCB, 0xBE, 0x39, 0x4A, 0x4C, 0x58, 0xCF, 0xD0, 0xEF, 0xAA, 0xFB, 0x43, 0x4D, 0x33, 0x85, 0x45, 0xF9, 0x02, 0x7F, 0x50, 0x3C, 0x9F, 0xA8, 0x51, 0xA3, 0x40, 0x8F, 0x92, 0x9D, 0x38, 0xF5, 0xBC, 0xB6, 0xDA, 0x21, 0x10, 0xFF, 0xF3, 0xD2, 0xCD, 0x0C, 0x13, 0xEC, 0x5F, 0x97, 0x44, 0x17, 0xC4, 0xA7, 0x7E, 0x3D, 0x64, 0x5D, 0x19, 0x73,

0x60, 0x81, 0x4F, 0xDC, 0x22, 0x2A, 0x90, 0x88, 0x46, 0xEE, 0xB8, 0x14, 0xDE, 0x5E, 0x0B, 0xDB, 0xE0, 0x32, 0x3A, 0x0A, 0x49, 0x06, 0x24, 0x5C, 0xC2, 0xD3, 0xAC, 0x62, 0x91, 0x95, 0xE4, 0x79, 0xE7, 0xC8, 0x37, 0x6D, 0x8D, 0xD5, 0x4E, 0xA9, 0x6C, 0x56, 0xF4, 0xEA, 0x65, 0x7A, 0xAE, 0x08, 0xBA, 0x78, 0x25, 0x2E, 0x1C, 0xA6, 0xB4, 0xC6, 0xE8, 0xDD, 0x74, 0x1F, 0x4B, 0xBD, 0x8B, 0x8A, 0x70, 0x3E, 0xB5, 0x66, 0x48, 0x03, 0xF6, 0x0E, 0x61, 0x35, 0x57, 0xB9, 0x86, 0xC1, 0x1D, 0x9E, 0xE1, 0xF8, 0x98, 0x11, 0x69, 0xD9, 0x8E, 0x94, 0x9B, 0x1E, 0x87, 0xE9, 0xCE, 0x55, 0x28, 0xDF, 0x8C, 0xA1, 0x89, 0x0D, 0xBF, 0xE6, 0x42, 0x68, 0x41, 0x99, 0x2D, 0x0F, 0xB0, 0x54, 0xBB, 0x16

};

Shift Rows

The second technique, Shift Rows, circular left shift a number of bytes equal to the row number. Figure 4 illustrates the process of shift rows. Sample code 1 shows the coded process.

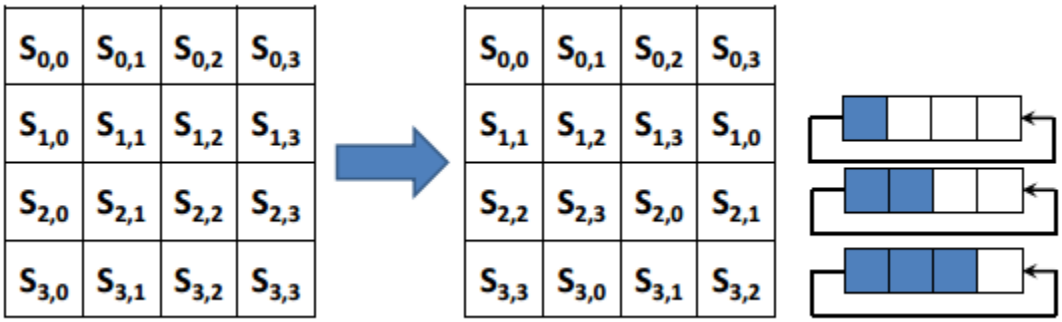

Figure 4: Shift row of the State

C code:

```
//sub-byte and shift rows
*(state_temp)=*(s+*(state));
*(state_temp+4)=*(s+*(state+4));
*(state_temp+8)=*(s+*(state+8));
*(state_temp+12)=*(s+*(state+12));
*(state_temp+1)=*(s+*(state+5));
*(state_temp+5)=*(s+*(state+9));
*(state_temp+9)=*(s+*(state+13));
*(state_temp+13)=*(s+*(state+1));
*(state_temp+2)=*(s+*(state+10));
*(state_temp+6)=*(s+*(state+14));
*(state_temp+10)=*(s+*(state+2));
*(state_temp+14)=*(s+*(state+6));
*(state_temp+3)=*(s+*(state+15));
*(state_temp+7)=*(s+*(state+3));
*(state_temp+11)=*(s+*(state+7));
*(state_temp+15)=*(s+*(state+11));
```

Sample code 1: Shift row of the State

Mix Columns

The third, Mix Columns, takes a column of data, multiplies it by a matrix, and stores the resulting column of data as shown in Figure 5. Each column of State is replaced by another column obtained by multiplying that column with a matrix in a particular field. The “mixing” helps diffuse the data. Sample code 2 shows the coded mixed columns.

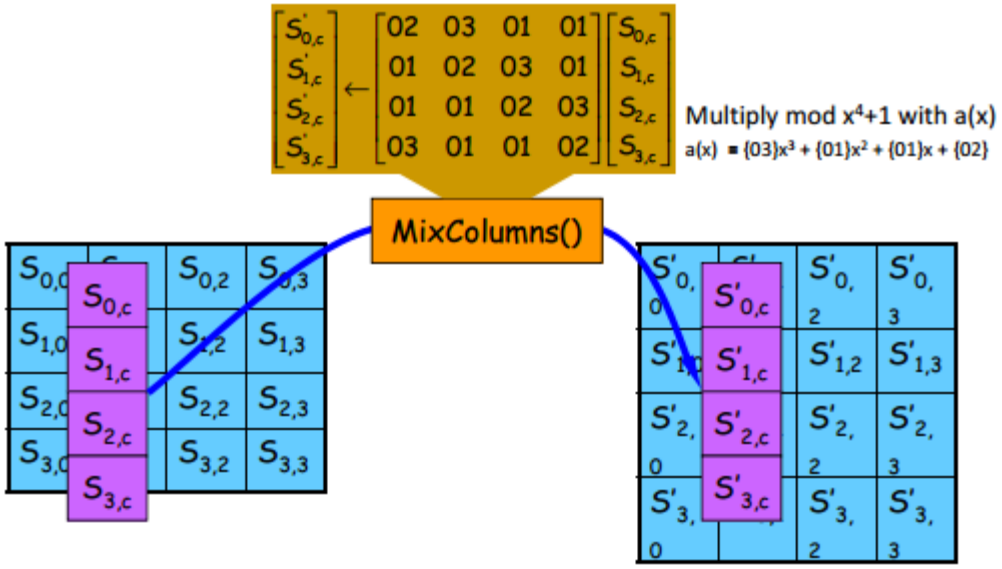

Figure 5: example of mix columns

C code:

```
//mix column
if(round_cnt<9){
int column, row, element;
for(column =0; column <4; column++){
for(row =0; row <4; row++){
for(element =0; element <4; element ++){
*(d_temp+element)=((*(state_temp+element+4*column)<<
((*(column_val+element+4*row)>>1)&0x01))*
((*(column_val+element+4*row)>>1)&0x01))
^(((*(state_temp+element+4*column)>>7)&
(0x01)&(*(column_val+element+4*row)>>1))*(0x1B))
^(((*(column_val+element+4*row))&(0x01))*
(*(state_temp+element+4*column)));
}
*(state+row+column*4)=(*(d_temp))^(*(d_temp+1))^
(*(d_temp+2))^(*(d_temp+3))^(*(cypher+row+column*4));
}
}
}
```

Sample code 2: Mix Columns

Add Round Key

The last step, called Add Round Key, performs an XOR operation of the data with corresponding values in the Round Key. Each block of input data is passed through the "main round" nine times for 128-bit encryption. The cipher key is user generated and, in conjunction with the Rijndael key schedule, generates the round keys. This process is shown in Figure 6 and coded in Sample code 3 (adding key) and 4 (generating key).

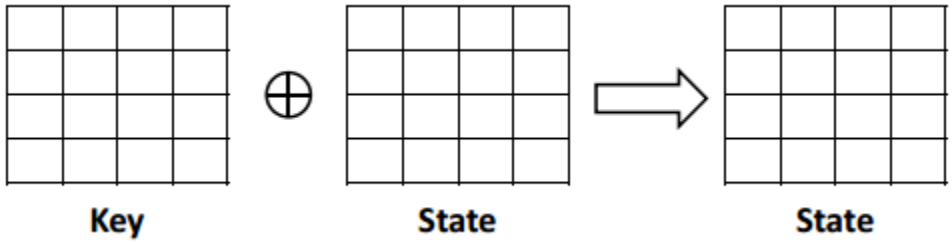

Figure 6: The process of Add Round Key

C code:

```
//add initial round key
voidadd_initial_rkey(uint8_t *state, uint8_t *cypher){
inti;
for(i=0;i<16;i++){
*(state+i)=*(state+i)^*(cypher+i);
}
}
```

Sample code 3: Adding round key

```
voidrkey_gen(uint8_t *cypher, uint8_t *s, uint8_t *Rcon,
 uint8_t round_cnt){
int k,i;
for(k =0; k <16; k++){
if(k <4){
if(k ==3){
*(d_temp+k)=(*(cypher+k))^(*(s+(*(cypher+12))))^(0x00);
}

elseif(k ==0){
*(d_temp+k)=(*(cypher+k))^(*(s+(*(cypher+k+13))))^
(*(Rcon+round_cnt));
}
else{
*(d_temp+k)=(*(cypher+k))^(*(s+(*(cypher+k+13))))^
(0x00);
}
}
else{
*(d_temp+k)=(*(d_temp+k-4))^(*(cypher+k));
}
}

for(i=0;i<16;i++){
*(cypher+i)=*(d_temp+i);
}
}
```

Sample code 4: Round key generator

AES Mix-Columns Transformation calculation

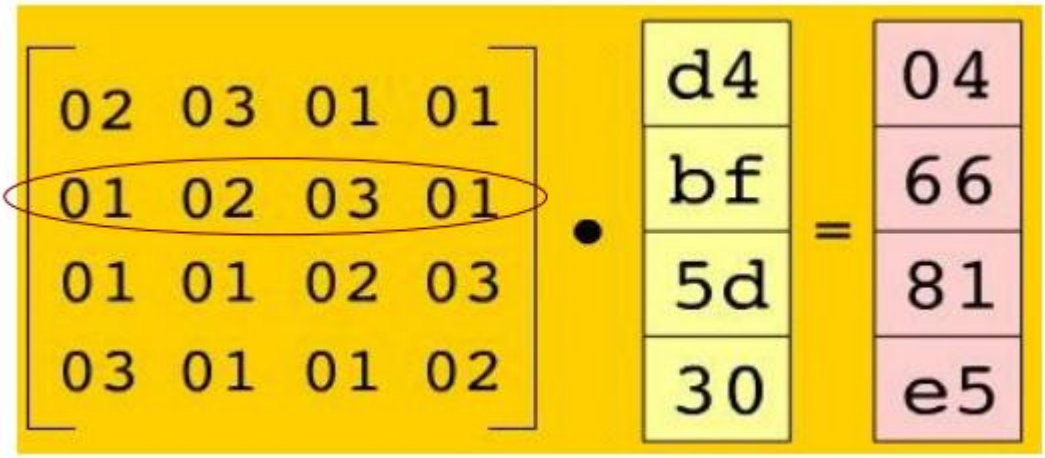

Figure 7: example matrix

r1 = {01.d4} + {02.bf} + {03.5d} + {01.30}

1. {02.bf}

{bf} . {02} = 1011 1111 << 1

= 0111 1110 XOR 0001 1011

= 0110 0101

2. {03.5d}

{5d} . {03} = {0101 1101 . 02} XOR { 0101 1101}

= 1011 1010 XOR 0101 1101

= 1110 0111

Therefore,

r1 = {01.d4} + {02.bf} + {03.5d} + {01.30}

= 1101 0100 XOR 0110 0101 XOR 1110 0111 XOR 0011 0000

= 0110 0110

= 66 (in Hex)

[5]

Sample inputs and outputs

Figure 8 illustrates an example data and cypher. In Figure 9 and 10, the 10 rounds of processes are shown.

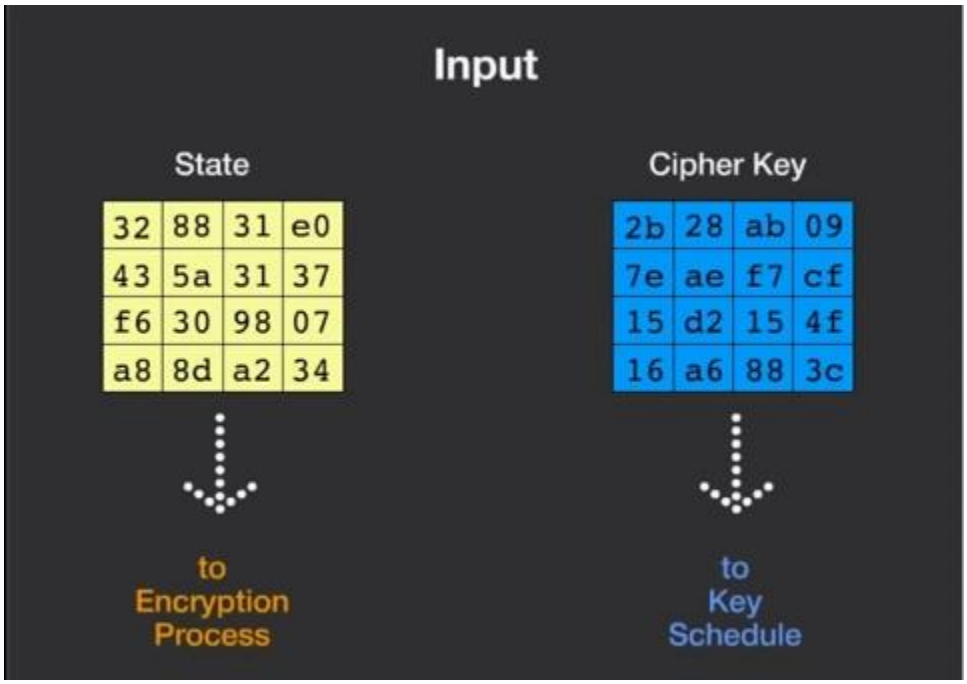

Figure 8: example Input of State and Cipher Key

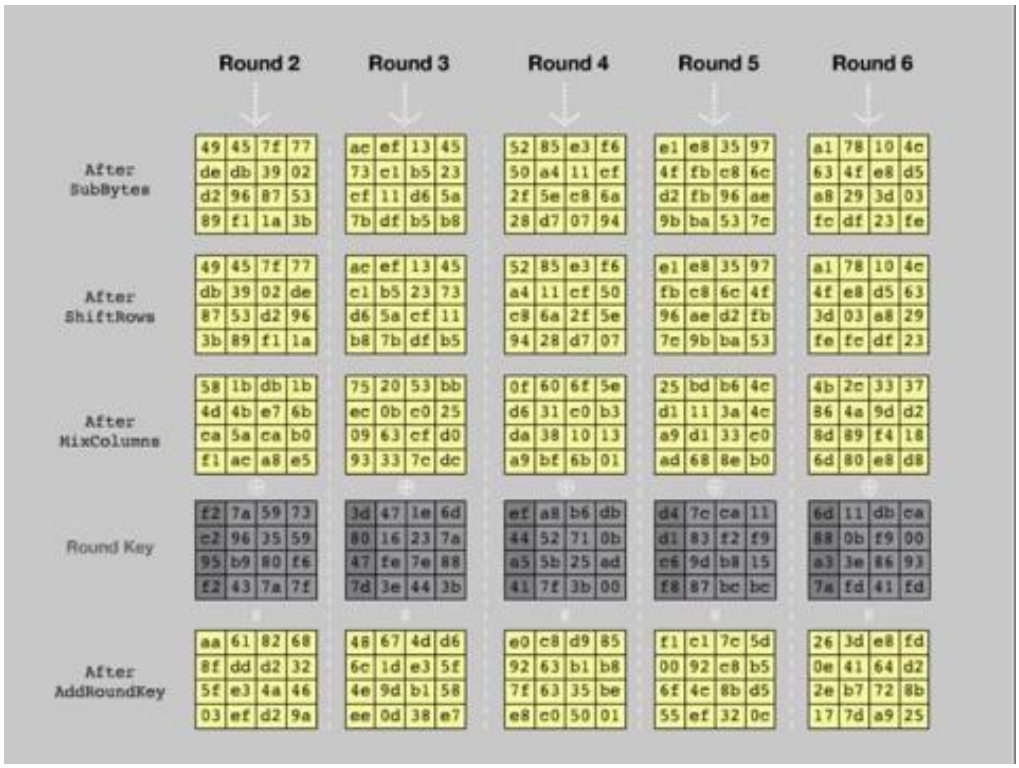

Figure 9: Round 2, 3, 4, 5, and 6 of the process

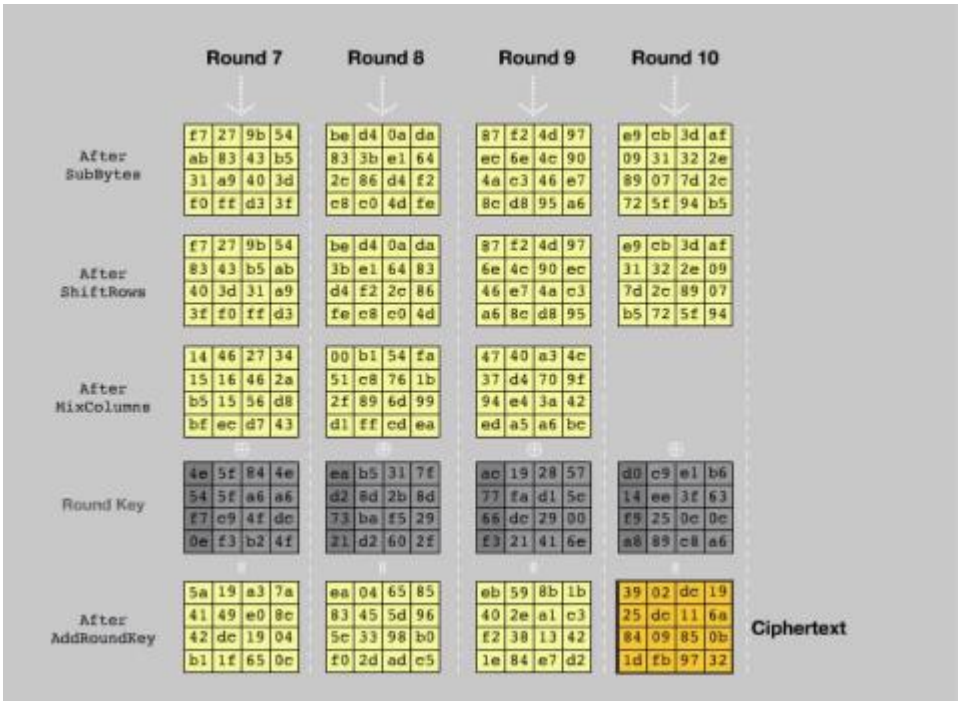

Figure 10: Round 7, 8, 9, and 10 of the process

Therefore, 32, 43 f6, a8, 88, 5a, 30, 8d, 31, 31, 98, a2, e0, 37, 07, 34 is the data, 2b, 7e, 15, 16, 28, ae, d2, a6, ab, f7, 15, 88, 09, cf, 4f, 3c is the cypher, and 39, 25, 84, 1d, 02, dc, 09, fb, dc, 11, 85, 97, 19, 6a, 0b, 32 is the expected ciphertext.

Display and testing results

A USART is used as an interface with a PC for display purposes. A USART (Universal Synchronous Asynchronous Receiver Transmitter) available on the AVR ATmega 328 is widely used for serial communication purposes. Sample code 5 shows the code used to initialize USART's send and receive features.

C code of USART initialization, send, and receive:

```
//initializing USART
voidusart_init(uint16_t baudin, uint32_t clk_speedin){
    uint32_t ubrr=(clk_speedin/16UL)/baudin-1;
    UBRR0H =(unsignedchar)(ubrr>>8);
    UBRR0L =(unsignedchar)ubrr;
/* Enable receiver and transmitter */
    UCSR0B =(1<<RXEN0)|(1<<TXEN0);
/* Set frame format: 8data, 1stop bit */
    UCSR0C =(1<<USBS0)|(3<<UCSZ00);
    UCSR0A &=~(1<<U2X0);
}

//send data through USART
voidusart_send( uint8_t data ){
/* Wait for empty transmit buffer */
while(!( UCSR0A &(1<<UDRE0)));
/* Put data into buffer, sends the data */
    UDR0 = data;
}

//receive data from USART
uint8_t usart_recv(void){
/* Wait for data to be received */
while(!(UCSR0A &(1<<RXC0)))
;
/* Get and return received data from buffer */
return UDR0;
}
```

Sample code 5: USART initialization, send, and receive

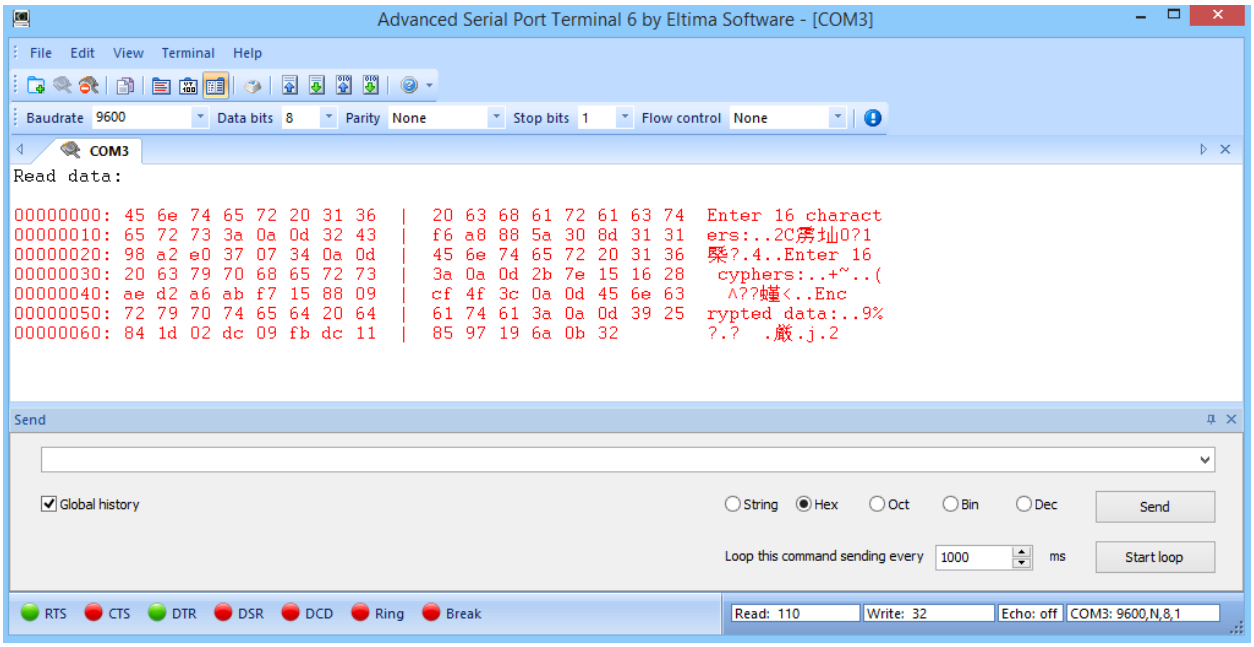

Figure 11: Advanced Serial Port terminal program

By using advanced port terminal 6, verification in terms of data, cypher, and final results are implemented. Figure 11 shows the final encrypted data which was 39, 25, 84, 1d, 02, dc, 09, fb, dc, 11, 85, 97, 19, 6a, 0b, 32

System integration

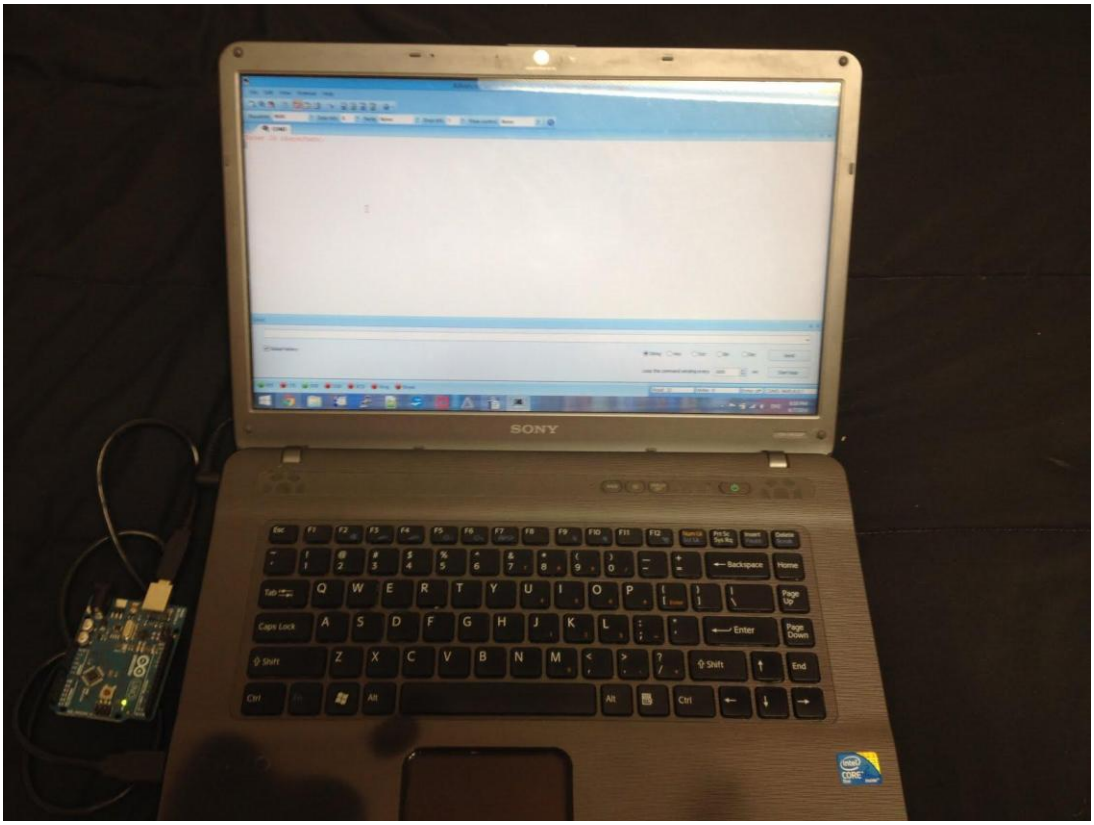

Figure 12: System integration of the project

In Figure 12, the system integration is shown. It consists of a laptop with Advanced serial port terminal program and an ATmega328 microcontroller.

Conclusion

The encryption was verified by using a terminal software to send data and cypher keys from a computer. With combination of the steps in the algorithm, the iteration and register usage were kept minimal. This project could be done by using an FPGA through VHDL. In the future, the use of USART in FPGA could be used to accomplish the same task. It is ready to be used as part of a large scale firmware to ensure data safety before transmitting. Overall, what was understood was the difference between using an FPGA and a microcontroller to implement the algorithm.

References

[1] Andrei Buruleanu, "Encryption Algorithm Implemented on FPGA", Digilent Contest Fourth Edition 2008. Web. 12 May 2014.

[2] Burr, W.E., "Selecting the Advanced Encryption Standard," *Security & Privacy, IEEE* , vol.1, no.2, pp.43,52, Mar-Apr 2003. 12 May 2014.

[3] Yang Xiao; Bo Sun; Hsiao-Hwa Chen; Guizani, S.; Ruhai Wang, "NIS05-1: Performance Analysis of Advanced Encryption Standard (AES)," *Global Telecommunications Conference, 2006. GLOBECOM '06.IEEE* , vol., no., pp.1,5, 27 November 2006. 12 May 2014.

[4] Bertoni, G.; Breveglieri, L.; Koren, I.; Maistri, P.; Piuri, V., "Detecting and locating faults in VLSI implementations of the Advanced Encryption Standard," *Defect and Fault Tolerance in VLSI Systems, 2003. Proceedings. 18th IEEE International Symposium on* , vol., no., pp.105,113, 3 November. 2003. 12 May 2014.

[5] Xintong, K. C. (n.d.). Understanding AES Mix-Columns Transformation Calculation. Wollongong, New South Wales, Australia.