**Angelica Semenec**

**012179180**

**Lab 2**

**Checksum Generation and Data Encryption/ Decryption**

**Purpose**

The purpose of this lab is to become familiar with two of the MSP432 on-board accelerators, the CRC-32 (cyclic redundancy check) checksum generator and the AES accelerator, which is used to encrypt and decrypt a block of data.

**Exercise 1: CRC-32**

**Exercise 1.1**

|  |
| --- |
| **crc32\_32-bit\_signature\_calculation.c** |
| #include <ti/devices/msp432p4xx/driverlib/driverlib.h>  #define CRC32\_POLY 0xEDB88320  #define CRC32\_INIT 0xFFFFFFFF  static uint8\_t myData[10240]; //Initialize size of myData to 10240  static uint32\_t calculateCRC32(uint8\_t\* data, uint32\_t length);  volatile uint32\_t hwCalculatedCRC, swCalculatedCRC;  //![Simple CRC32 Example]  int main(void)  {  uint32\_t i;  for (i = 0; i < 10240; i++) //Iterates through each index (i) of myData  {  do  {  myData[i] = rand(); //Sets myData[i] to a random value  }while(myData[i]%2 != i%2); //Checks if myData and index are odd or even  //Repeats the loop if both are not odd or both are not even  printf(" %u", myData[i]); //Prints the contents of myData[i]  }  .  .  .  } |

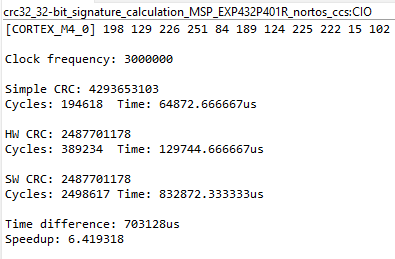
The array size is set to 10240. When *main()* is called, the for loop iterates through each index of the array *myData[]*. Each element is set to a random value using the *rand()* function. Each index value of the array follows a regularly repeating pattern that alternates between even and odd values at each step. The modulus function ( *%* ) is used to check the parity of *myData[i]* and index *i* by checking the least significant bit of the numbers. If the values are the same, then the values of *myData[]* will follow the same repeating parity pattern as that of the index. If the values differ, then the loop is repeated until they are the same. The do-while loop is used to ensure that the loop is executed at least once. The contents of *myData[]* is printed to verify the functionality of the program.

**Exercise 1.2**

|  |
| --- |
| **crc32\_32-bit\_signature\_calculation.c** |
| /\* Simple Checksum \*/  static uint32\_t compute\_simple\_checksum(uint8\_t\* data, uint32\_t length)  {  uint32\_t ii;  uint32\_t checksum = 0; //Initializes checksum to 0  for(ii=0;ii<length;ii++) //Increments through each element of the data array  {  checksum = checksum + data[ii]; //Adds each element of the data array  }  return ~checksum; //Returns bit-reversed checksum  } |

**Exercise 1.3**

|  |
| --- |
| **crc32\_32-bit\_signature\_calculation.c** |
| int main(void)  {  /\* Initialize timing variables \*/  uint32\_t t0, t1, tHW, tSW, tSim;  double tHW\_us, tSW\_us, tSim\_us;  /\* Setup a timer \*/  MAP\_Timer32\_initModule(TIMER32\_0\_BASE, TIMER32\_PRESCALER\_1,  TIMER32\_32BIT, TIMER32\_FREE\_RUN\_MODE);  MAP\_Timer32\_startTimer(TIMER32\_0\_BASE, 0);  uint32\_t clk\_freq = MAP\_CS\_getMCLK(); //Gets the clock frequency  printf("Clock frequency: %u\n\n", clk\_freq);  t0 = MAP\_Timer32\_getValue(TIMER32\_0\_BASE);  MAP\_CRC32\_setSeed(CRC32\_INIT, CRC32\_MODE);  for (ii = 0; ii < 10240; ii++) //ii updated to 10240  MAP\_CRC32\_set8BitData(myData[ii], CRC32\_MODE);  /\* Getting the result from the hardware module \*/  hwCalculatedCRC = MAP\_CRC32\_getResultReversed(CRC32\_MODE) ^ 0xFFFFFFFF;  t1 = MAP\_Timer32\_getValue(TIMER32\_0\_BASE);  tHW = t0 - t1;  t0 = MAP\_Timer32\_getValue(TIMER32\_0\_BASE);  /\* Calculating the CRC32 checksum through software \*/  swCalculatedCRC = calculateCRC32((uint8\_t\*) myData, 10240); //length updated to 10240  t1 = MAP\_Timer32\_getValue(TIMER32\_0\_BASE);  tSW = t0 - t1;  t0 = MAP\_Timer32\_getValue(TIMER32\_0\_BASE);  /\* Calculating the checksum through addition \*/  simpleSum = compute\_simple\_checksum((uint8\_t\*) myData, 10240);  t1 = MAP\_Timer32\_getValue(TIMER32\_0\_BASE);  tSim = t0 - t1;  /\* Calculate time in microseconds for each checksum computation \*/  tSim\_us = ((double)tSim/(double)clk\_freq)\*1000000;  tHW\_us = ((double)tHW/(double)clk\_freq)\*1000000;  tSW\_us = ((double)tSW/(double)clk\_freq)\*1000000;  /\* Print Information \*/  printf("Simple sum: %u\n", simpleSum);  printf("Cycles: %u\tTime: %fus\n\n", tSim, tSim\_us);  printf("HW CRC: %u\n", hwCalculatedCRC);  printf("Cycles: %u\tTime: %fus\n\n", tHW, tHW\_us);  printf("SW CRC: %u\n", swCalculatedCRC);  printf("Cycles: %u\tTime: %fus\n\n", tSW, tSW\_us);  printf("Time difference: %dus\nSpeedup: %f", ((int)tSW\_us - (int)tHW\_us), (tSW\_us/tHW\_us));  .  .  .  } |



**Exercise 1.4**

|  |
| --- |
| **crc32\_32-bit\_signature\_calculation.c** |
| int main(void)  {  .  .  .  /\* Print values before LSB is changed \*/  printf("myData[20]: %u myData[21]: %u\n", myData[20], myData[21]);  uint32\_t hw20, sim20, hw21, sim21;  /\* Get HW and Simple checksum when myData[20] is modified \*/  myData[20] = myData[20] ^ 1;  MAP\_CRC32\_setSeed(CRC32\_INIT, CRC32\_MODE);  for (ii = 0; ii < 10240; ii++)  MAP\_CRC32\_set8BitData(myData[ii], CRC32\_MODE);  /\* Getting results \*/  hw20 = MAP\_CRC32\_getResultReversed(CRC32\_MODE) ^ 0xFFFFFFFF;  sim20 = compute\_simple\_checksum((uint8\_t\*) myData, 10240);  /\* Get HW and Simple checksum when myData[21] is additionally modified \*/  myData[21] = myData[21] ^ 1;  MAP\_CRC32\_setSeed(CRC32\_INIT, CRC32\_MODE);  for (ii = 0; ii < 10240; ii++)  MAP\_CRC32\_set8BitData(myData[ii], CRC32\_MODE);  /\* Getting results \*/  hw21 = MAP\_CRC32\_getResultReversed(CRC32\_MODE) ^ 0xFFFFFFFF;  sim21 = compute\_simple\_checksum((uint8\_t\*) myData, 10240);  printf("LSB change %u %u\n\n", myData[20], myData[21]);  printf("No change\nHW: %u\tSimple: %u\n\n", hwCalculatedCRC, simpleCRC);  printf("myData[20] LSB change\nHW: %u\tSimple: %u\n\n", hw20, sim20);  printf("myData[21] LSB change\nHW: %u\tSimple: %u\n\n", hw21, sim21);  .  .  .  } |

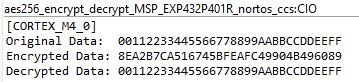
After the value of myData[20] is changed by modifying the LSB, both methods of calculating the checksum change in value. When the value of myData[21] is additionally modified by changing the LSB, the value of the HW calculated checksum is different from the original checksum value (no modifications) and the checksum value when myData[20] was modified. Each change in the data produced a different checksum value when utilizing the HW calculated method. For the simple checksum, the checksum value was the same as the original value before any modifications were made. When the LSB on an odd number is modified, the value decreases by 1. When the LSB of an even value is modified, the value increases by 1. For the simple checksum, which is calculated through adding all of the array elements together, the modifications to the checksum value cancel each other out in value and so the checksum remains unchanged from its original value even though much of the data may have been modified.

The HW calculated checksum is a more reliable method of detecting data modifications because it is more likely to produce a unique checksum than the simple method, which cannot detect certain data modifications such as the one described by the example above.

**Exercise 2: AES**

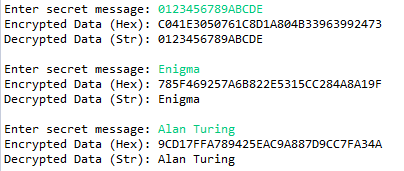
**Exercise 2.1**

|  |
| --- |
| **aes256\_encrypt\_decrypt.c** |
| int main(void)  {  .  .  .  /\* Printing data, encrypted data, and decrypted data \*/  int ii, jj;  for (jj = 0; jj < 3; jj++)  {  if(jj == 0)  {  printf("\nOriginal Data: ");  }  else if(jj == 1)  {  printf("Encrypted Data: ");  }  else if(jj == 2)  {  printf("Decrypted Data: ");  }  for (ii = 0; ii < 16; ii++)  {  if(jj == 0)  printf("%02X", Data[ii]);  else if(jj == 1)  printf("%02X", DataAESencrypted[ii]);  else if(jj == 2)  printf("%02X", DataAESdecrypted[ii]);  }  printf("\n");  }  .  .  .  } |



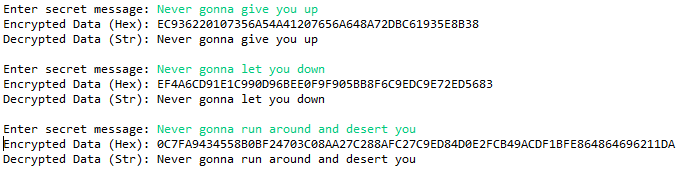
**Exercise 2.2**

|  |
| --- |
| **aes256\_encrypt\_decrypt.c** |
| int main(void)  {  .  .  .  char txt[16];  /\* Get user input text message \*/  while(1)  {  printf("Enter secret message: ");  gets(txt);  /\* Encrypt and decrypt message \*/  encrypt\_message\_16(txt, DataAESencrypted, CipherKey);  decrypt\_message\_16(DataAESencrypted, DataAESdecrypted, CipherKey);  printf("Encrypted Data (Hex): ");  int ii;  for (ii = 0; ii < 16; ii++)  {  printf("%02X", DataAESencrypted[ii]);  }  printf("\nDecrypted Data (Str): %s\n\n", DataAESdecrypted);  }  .  .  .  }  /\* Function to encrypt 16-Byte string message \*/  void encrypt\_message\_16(char\* str, uint8\_t\* encrypted, uint8\_t\* key)  {  /\* Load a cipher key to module \*/  MAP\_AES256\_setCipherKey(AES256\_BASE, key, AES256\_KEYLENGTH\_256BIT);  /\* Encrypt data with preloaded cipher key \*/  MAP\_AES256\_encryptData(AES256\_BASE, str, encrypted);  }  /\* Function to decrypt encrypted 16-Byte string message \*/  void decrypt\_message\_16(uint8\_t\* data, uint8\_t\* decrypted, uint8\_t\* key)  {  /\* Load a decipher key to module \*/  MAP\_AES256\_setDecipherKey(AES256\_BASE, key, AES256\_KEYLENGTH\_256BIT);  /\* Decrypt data with keys that were generated during encryption - takes  214 MCLK cyles. This function will generate all round keys needed for  decryption first and then the encryption process starts \*/  MAP\_AES256\_decryptData(AES256\_BASE, data, decrypted);  } |



**Exercise 2.3**

|  |
| --- |
| **aes256\_encrypt\_decrypt.c** |
| int main(void)  {  .  .  .  char txt[1600];  int str\_length;  /\* Get user input text message \*/  while(1)  {  printf("Enter secret message: "); //User prompt  gets(txt);  str\_length = strlen(txt); //Gets the length of the string  /\* Encrypt and decrypt the message \*/  encrypt\_message(txt, DataAESencrypted, CipherKey);  decrypt\_message(DataAESencrypted, str\_length, DataAESdecrypted, CipherKey);  printf("Encrypted Data (Hex): "); //Prints encrypted data in hex format  int ii;  for (ii = 0; ii < str\_length; ii++)  {  printf("%02X", DataAESencrypted[ii]);  }  printf("\nDecrypted Data (Str): %s\n\n", DataAESdecrypted); //Prints the decrypted string message  }  .  .  .  }  void encrypt\_message(char\* str, uint8\_t\* encrypted, uint8\_t\* key)  {  /\* Load a cipher key to module \*/  MAP\_AES256\_setCipherKey(AES256\_BASE, key, AES256\_KEYLENGTH\_256BIT);  int ii;  int loop\_count = (strlen(str)/16) + (strlen(str)%16 != 0); //Gets the number of loop iterations  for (ii = 0; ii < loop\_count; ii++) //Encrypts 16-Byte Blocks of data  MAP\_AES256\_encryptData(AES256\_BASE, str + ii\*16, encrypted + ii\*16); //Encrypts data  }  void decrypt\_message(uint8\_t\* data, int data\_length, uint8\_t\* decrypted, uint8\_t\* key)  {  /\* Load a decipher key to module \*/  MAP\_AES256\_setDecipherKey(AES256\_BASE, key, AES256\_KEYLENGTH\_256BIT);  int ii;  int loop\_count = (data\_length/16) + (data\_length%16 != 0); //Gets the number of loop iterations  for (ii = 0; ii < loop\_count; ii++) //Decrypts 16-Byte blocks of data  MAP\_AES256\_decryptData(AES256\_BASE, data + (ii\*16), decrypted + (ii\*16)); //Decrypts data  } |



**Appendix**

**Exercise 1**

|  |
| --- |
| **crc32\_32-bit\_signature\_calculation.c** |
| #include <ti/devices/msp432p4xx/driverlib/driverlib.h>  #define CRC32\_POLY 0xEDB88320  #define CRC32\_INIT 0xFFFFFFFF  static uint8\_t myData[10240]; //Initialize size of myData to 10240  static uint32\_t calculateCRC32(uint8\_t\* data, uint32\_t length);  static uint32\_t compute\_simple\_checksum(uint8\_t\* data, uint32\_t length);  volatile uint32\_t hwCalculatedCRC, swCalculatedCRC, simpleSum;  //![Simple CRC32 Example]  int main(void)  {  uint32\_t i;  for (i = 0; i < 10240; i++) //Iterates through each index (i) of myData  {  do  {  myData[i] = rand(); //Sets myData[i] to a random value  }while(myData[i]%2 != i%2); //Checks if myData and index are odd or even  //Repeats the loop if both are not odd or both are not even  }  for (i = 0; i < 25; i++)  {  printf("%u ", myData[i]); //Prints the contents of myData[i]  }  printf("\n\n");  uint32\_t ii;  /\* Stop WDT \*/  MAP\_WDT\_A\_holdTimer();  /\* Initialize timing variables \*/  uint32\_t t0, t1, tHW, tSW, tSim;  double tHW\_us, tSW\_us, tSim\_us;  /\* Setup a timer \*/  MAP\_Timer32\_initModule(TIMER32\_0\_BASE, TIMER32\_PRESCALER\_1,  TIMER32\_32BIT, TIMER32\_FREE\_RUN\_MODE);  MAP\_Timer32\_startTimer(TIMER32\_0\_BASE, 0);  uint32\_t clk\_freq = MAP\_CS\_getMCLK(); //Gets the clock frequency  printf("Clock frequency: %u\n\n", clk\_freq);  t0 = MAP\_Timer32\_getValue(TIMER32\_0\_BASE);  MAP\_CRC32\_setSeed(CRC32\_INIT, CRC32\_MODE);  for (ii = 0; ii < 10240; ii++) //ii updated to 10240  MAP\_CRC32\_set8BitData(myData[ii], CRC32\_MODE);  /\* Getting the result from the hardware module \*/  hwCalculatedCRC = MAP\_CRC32\_getResultReversed(CRC32\_MODE) ^ 0xFFFFFFFF;  t1 = MAP\_Timer32\_getValue(TIMER32\_0\_BASE);  tHW = t0 - t1;  t0 = MAP\_Timer32\_getValue(TIMER32\_0\_BASE);  /\* Calculating the CRC32 checksum through software \*/  swCalculatedCRC = calculateCRC32((uint8\_t\*) myData, 10240); //length updated to 10240  t1 = MAP\_Timer32\_getValue(TIMER32\_0\_BASE);  tSW = t0 - t1;  t0 = MAP\_Timer32\_getValue(TIMER32\_0\_BASE);  /\* Calculating the checksum through addition \*/  simpleSum = compute\_simple\_checksum((uint8\_t\*) myData, 10240);  t1 = MAP\_Timer32\_getValue(TIMER32\_0\_BASE);  tSim = t0 - t1;  /\* Calculate time in microseconds for each checksum computation \*/  tSim\_us = ((double)tSim/(double)clk\_freq)\*1000000;  tHW\_us = ((double)tHW/(double)clk\_freq)\*1000000;  tSW\_us = ((double)tSW/(double)clk\_freq)\*1000000;  /\* Print Information \*/  printf("Simple Sum: %u\n", simpleSum);  printf("Cycles: %u\tTime: %fus\n\n", tSim, tSim\_us);  printf("HW CRC: %u\n", hwCalculatedCRC);  printf("Cycles: %u\tTime: %fus\n\n", tHW, tHW\_us);  printf("SW CRC: %u\n", swCalculatedCRC);  printf("Cycles: %u\tTime: %fus\n\n", tSW, tSW\_us);  printf("Time difference: %dus\nSpeedup: %f\n\n", ((int)tSW\_us - (int)tHW\_us), (tSW\_us/tHW\_us));  /\* Print values before LSB is changed \*/  printf("myData[20]: %u myData[21]: %u\n", myData[20], myData[21]);  uint32\_t hw20, sim20, hw21, sim21;  /\* Get HW and Simple checksum when myData[20] is modified \*/  myData[20] = myData[20] ^ 1;  MAP\_CRC32\_setSeed(CRC32\_INIT, CRC32\_MODE);  for (ii = 0; ii < 10240; ii++)  MAP\_CRC32\_set8BitData(myData[ii], CRC32\_MODE);  /\* Getting results \*/  hw20 = MAP\_CRC32\_getResultReversed(CRC32\_MODE) ^ 0xFFFFFFFF;  sim20 = compute\_simple\_checksum((uint8\_t\*) myData, 10240);  /\* Get HW and Simple checksum when myData[21] is additionally modified \*/  myData[21] = myData[21] ^ 1;  MAP\_CRC32\_setSeed(CRC32\_INIT, CRC32\_MODE);  for (ii = 0; ii < 10240; ii++)  MAP\_CRC32\_set8BitData(myData[ii], CRC32\_MODE);  /\* Getting results \*/  hw21 = MAP\_CRC32\_getResultReversed(CRC32\_MODE) ^ 0xFFFFFFFF;  sim21 = compute\_simple\_checksum((uint8\_t\*) myData, 10240);  printf("LSB change %u %u\n\n", myData[20], myData[21]);  printf("No change\nHW: %u\tSimple: %u\n\n", hwCalculatedCRC, simpleCRC);  printf("myData[20] LSB change\nHW: %u\tSimple: %u\n\n", hw20, sim20);  printf("myData[21] LSB change\nHW: %u\tSimple: %u\n\n", hw21, sim21);  /\* Pause for the debugger \*/  \_\_no\_operation();  }  //![Simple CRC32 Example]  /\* Standard software calculation of CRC32 \*/  static uint32\_t calculateCRC32(uint8\_t\* data, uint32\_t length)  {  uint32\_t ii, jj, byte, crc, mask;;  crc = 0xFFFFFFFF;  for(ii=0;ii<length;ii++)  {  byte = data[ii];  crc = crc ^ byte;  for (jj = 0; jj < 8; jj++)  {  mask = -(crc & 1);  crc = (crc >> 1) ^ (CRC32\_POLY & mask);  }  }  return ~crc;  }  /\* Simple Checksum \*/  static uint32\_t compute\_simple\_checksum(uint8\_t\* data, uint32\_t length)  {  uint32\_t ii;  uint32\_t checksum = 0; //Initializes checksum to 0  for(ii=0;ii<length;ii++) //Increments through each element of the data array  {  checksum = checksum + data[ii]; //Adds each element of the data array  }  return ~checksum; //Returns bit-reversed checksum  } |

**Exercise 2**

**Exercise 2.1**

|  |
| --- |
| **aes256\_encrypt\_decrypt.c** |
| /\* DriverLib Includes \*/  #include <ti/devices/msp432p4xx/driverlib/driverlib.h>  /\* Standard Includes \*/  #include <stdint.h>  #include <stdbool.h>  /\* Statics \*/  static uint8\_t Data[16] =  { 0x00, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88, 0x99, 0xaa, 0xbb, 0xcc,  0xdd, 0xee, 0xff };  static uint8\_t CipherKey[32] =  { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c,  0x0d, 0x0e, 0x0f, 0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17, 0x18,  0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f };  static uint8\_t DataAESencrypted[16]; // Encrypted data  static uint8\_t DataAESdecrypted[16]; // Decrypted data  int main(void)  {  /\* Stop Watchdog \*/  MAP\_WDT\_A\_holdTimer();  //![Simple AES256 Example]  /\* Load a cipher key to module \*/  MAP\_AES256\_setCipherKey(AES256\_BASE, CipherKey, AES256\_KEYLENGTH\_256BIT);  /\* Encrypt data with preloaded cipher key \*/  MAP\_AES256\_encryptData(AES256\_BASE, Data, DataAESencrypted);  /\* Load a decipher key to module \*/  MAP\_AES256\_setDecipherKey(AES256\_BASE, CipherKey,AES256\_KEYLENGTH\_256BIT);  /\* Decrypt data with keys that were generated during encryption - takes  214 MCLK cyles. This function will generate all round keys needed for  decryption first and then the encryption process starts \*/  MAP\_AES256\_decryptData(AES256\_BASE, DataAESencrypted, DataAESdecrypted);  //![Simple AES256 Example]    /\* Printing data, encrypted data, and decrypted data \*/  int ii, jj;  for (jj = 0; jj < 3; jj++)  {  if(jj == 0)  {  printf("\nOriginal Data: ");  }  else if(jj == 1)  {  printf("Encrypted Data: ");  }  else if(jj == 2)  {  printf("Decrypted Data: ");  }  for (ii = 0; ii < 16; ii++)  {  if(jj == 0)  printf("%02X", Data[ii]);  else if(jj == 1)  printf("%02X", DataAESencrypted[ii]);  else if(jj == 2)  printf("%02X", DataAESdecrypted[ii]);  }  printf("\n");  }  /\* Array DataAESdecrypted should now contain the same data as array Data \*/  while(1)  {  MAP\_PCM\_gotoLPM0();  }  } |

**Exercise 2.2**

|  |
| --- |
| **aes256\_encrypt\_decrypt.c** |
| /\* DriverLib Includes \*/  #include <ti/devices/msp432p4xx/driverlib/driverlib.h>  /\* Standard Includes \*/  #include <stdint.h>  #include <stdbool.h>  /\* Statics \*/  static uint8\_t CipherKey[32] =  { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c,  0x0d, 0x0e, 0x0f, 0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17, 0x18,  0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f };  static uint8\_t DataAESencrypted[16]; // Encrypted data  static uint8\_t DataAESdecrypted[16]; // Decrypted data  /\* New Function Declarations \*/  void encrypt\_message\_16(char\* str, uint8\_t\* encrypted, uint8\_t\* key);  void decrypt\_message\_16(uint8\_t\* data, uint8\_t\* decrypted, uint8\_t\* key);  int main(void)  {  /\* Stop Watchdog \*/  MAP\_WDT\_A\_holdTimer();  char txt[16];  /\* Get user input text message \*/  while(1)  {  printf("Enter secret message: ");  gets(txt);  /\* Encrypt and decrypt the message \*/  encrypt\_message\_16(txt, DataAESencrypted, CipherKey);  decrypt\_message\_16(DataAESencrypted, DataAESdecrypted, CipherKey);  printf("Encrypted Data (Hex): ");  int ii;  for (ii = 0; ii < 16; ii++)  {  printf("%02X", DataAESencrypted[ii]);  }  printf("\nDecrypted Data (Str): %s\n\n", DataAESdecrypted);  }  /\* Array DataAESdecrypted should now contain the same data as array Data \*/  while(1)  {  MAP\_PCM\_gotoLPM0();  }  }  /\* Function to encrypt 16-Byte string message \*/  void encrypt\_message\_16(char\* str, uint8\_t\* encrypted, uint8\_t\* key)  {  /\* Load a cipher key to module \*/  MAP\_AES256\_setCipherKey(AES256\_BASE, key, AES256\_KEYLENGTH\_256BIT);  /\* Encrypt data with preloaded cipher key \*/  MAP\_AES256\_encryptData(AES256\_BASE, str, encrypted);  }  /\* Function to decrypt encrypted 16-Byte string message \*/  void decrypt\_message\_16(uint8\_t\* data, uint8\_t\* decrypted, uint8\_t\* key)  {  /\* Load a decipher key to module \*/  MAP\_AES256\_setDecipherKey(AES256\_BASE, key, AES256\_KEYLENGTH\_256BIT);  /\* Decrypt data with keys that were generated during encryption - takes  214 MCLK cyles. This function will generate all round keys needed for  decryption first and then the encryption process starts \*/  MAP\_AES256\_decryptData(AES256\_BASE, data, decrypted);  } |

**Exercise 2.3**

|  |
| --- |
| **aes256\_encrypt\_decrypt.c** |
| /\* DriverLib Includes \*/  #include <ti/devices/msp432p4xx/driverlib/driverlib.h>  /\* Standard Includes \*/  #include <stdint.h>  #include <stdbool.h>  #include <string.h>  /\* Statics \*/  static uint8\_t CipherKey[32] =  { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c,  0x0d, 0x0e, 0x0f, 0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17, 0x18,  0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f };  static uint8\_t DataAESencrypted[1600]; // Encrypted data  static uint8\_t DataAESdecrypted[1600]; // Decrypted data  /\* Function Declarations \*/  void encrypt\_message(char\* str, uint8\_t\* encrypted, uint8\_t\* key);  void decrypt\_message(uint8\_t\* data, int data\_length, uint8\_t\* decrypted, uint8\_t\* key);  int main(void)  {  /\* Stop Watchdog \*/  MAP\_WDT\_A\_holdTimer();  char txt[1600];  int str\_length;  /\* Get user input text message \*/  while(1)  {  printf("Enter secret message: "); //User prompt  gets(txt);  str\_length = strlen(txt); //Gets the length of the string  /\* Encrypt and decrypt the message \*/  encrypt\_message(txt, DataAESencrypted, CipherKey);  decrypt\_message(DataAESencrypted, str\_length, DataAESdecrypted, CipherKey);  printf("Encrypted Data (Hex): "); //Prints encrypted data in hex format  int ii;  for (ii = 0; ii < str\_length; ii++)  {  printf("%02X", DataAESencrypted[ii]);  }  printf("\nDecrypted Data (Str): %s\n\n", DataAESdecrypted); //Prints the decrypted string message  }    /\* Array DataAESdecrypted should now contain the same data as array Data \*/  while(1)  {  MAP\_PCM\_gotoLPM0();  }  }  void encrypt\_message(char\* str, uint8\_t\* encrypted, uint8\_t\* key)  {  /\* Load a cipher key to module \*/  MAP\_AES256\_setCipherKey(AES256\_BASE, key, AES256\_KEYLENGTH\_256BIT);  int ii;  int loop\_count = (strlen(str)/16) + (strlen(str)%16 != 0); //Gets the number of loop iterations  for (ii = 0; ii < loop\_count; ii++) //Encrypts 16-Byte Blocks of data  MAP\_AES256\_encryptData(AES256\_BASE, str + ii\*16, encrypted + ii\*16); //Encrypts data  }  void decrypt\_message(uint8\_t\* data, int data\_length, uint8\_t\* decrypted, uint8\_t\* key)  {  /\* Load a decipher key to module \*/  MAP\_AES256\_setDecipherKey(AES256\_BASE, key, AES256\_KEYLENGTH\_256BIT);  int ii;  int loop\_count = (data\_length/16) + (data\_length%16 != 0); //Gets the number of loop iterations  for (ii = 0; ii < loop\_count; ii++) //Decrypts 16-Byte blocks of data  MAP\_AES256\_decryptData(AES256\_BASE, data + (ii\*16), decrypted + (ii\*16)); //Decrypts data  } |