Implementing Security in a Personal Security Device

by

Priyansha Gupta

Implementing Security in a Personal Security Device

Presented to the Faculty of the Graduate School of

The University of California Los Angeles

in Partial Fulfillment

of the Requirements

for the Degree of

Masters of Science in Engineering

The University of California Los Angeles

December, 2013

Dedication

To my beloved husband Gaurav and to my parents

Acknowledgements

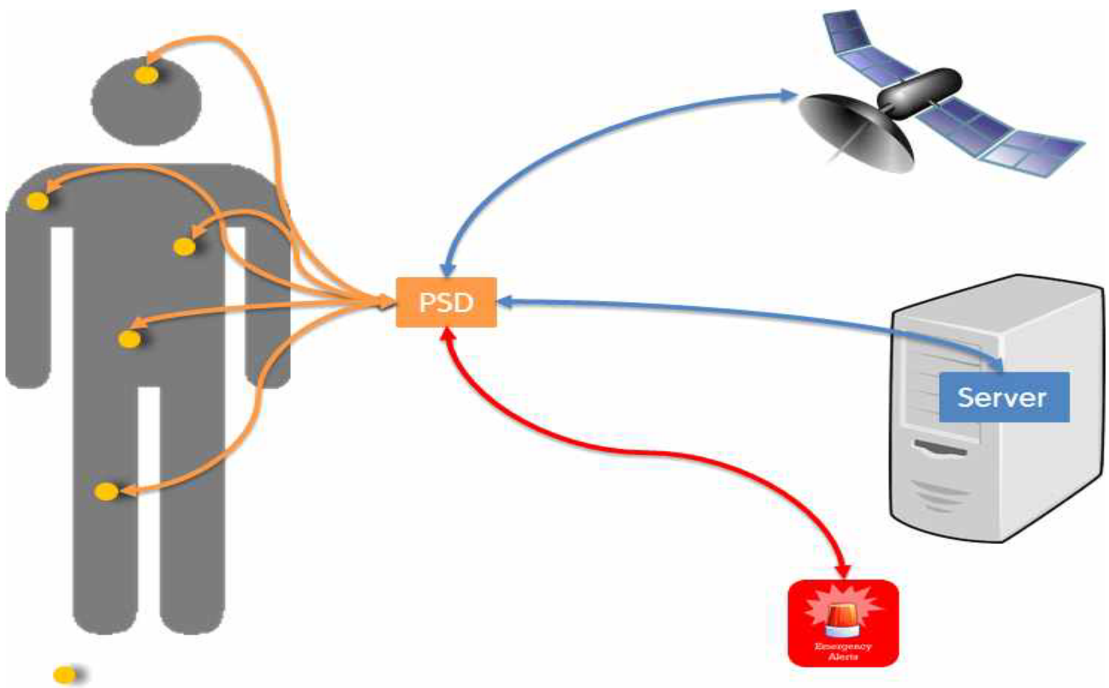
It has been an enlightening experience to work under the guidance of Professor Peter Reiher. I sincerely thank him for his keen insight and continuous encouragement throughout the course of my study.

I would also like to acknowledge my colleague, Jong Hyun Lee, for his support and encouragement.

Table of Contents

INTRODUCTION

Wireless medical devices provide a multitude of benefits for both patients and physicians. These benefits include increasing patient mobility without the need to be in a hospital bed and providing the ability of physicians to remotely access and monitor patient data regardless of the location of the patient or physician. This technology greatly enhances patient outcomes by allowing physicians access to real-time data on patients without the physical restraints of being in the same location [2].



<Fig1: Conceptual design of PSD>

The internet connected devices increase connectivity and provide greater functionality, however, they also increase risks of both unintentional and malicious tampering of PHI over a multitude of wireless signals and data from medical devices. The FDA encourages wireless encryption to protect against unauthorized wireless access to device data [2].

The idea of implementing secure communication in the medical device itself has been around but using a separate Personal Security Device (PSD) as an intermediary offers several advantages [9]. In essence, PSD works by providing an alternate secure communication path between the medical device and access point (AP), in addition to the regular communication between them. Some changes are required in the AP to handle both secured and unsecured communication. The secured communication can serve as an authentication for the unsecured communication thereby minimizing the risk of security attacks.

The PSD concept

- requires no changes to the medical device (hardware and software),
- provides unlimited access in emergency situation by just turning off the PSD,
- reduces the burden on medical device by shifting security responsibility to PSD

The aim of this project is to determine if cryptographic software can be implemented in commercially available hardware like Arduino that have limited amount of memory. More specifically, the different amounts of memory (eg Flash, EEPROM, SRAM) used and remaining need to be determined. For the scope of this project, Advanced Encryption Standard (AES) was considered for the Arduino Mega 2560 platform.

Due to this social and technical demand, this project will aim to implement security in a device that can collect information (from all available and existing medical devices) and communicate with server. This device aims to attain the security feature in the firmware of a device with memory constraints.

BACKGROUND KNOWLEDGE

1. Microcontroller board: Arduino

Microprocessor board is a tool for making computers that can sense and control more of the physical world than the desktop computer [1]. Arduino is an open-source physical computing platform based on a simple microcontroller board, and an environment for writing software for the board. This microprocessor board can be used to develop interactive objects by taking inputs from a variety of switches or sensors and controlling a variety of lights, motors, and other physical outputs. Arduino projects can be stand-alone or they can communicate with software running on another computer. The boards can be assembled by hand or purchased preassembled.

Arduino simplifies the process of working with microcontrollers andit offers some advantage for teachers, students, and interested amateurs over other systems:

i) Inexpensive: Arduino boards are relatively inexpensive compared to other microcontroller platforms.

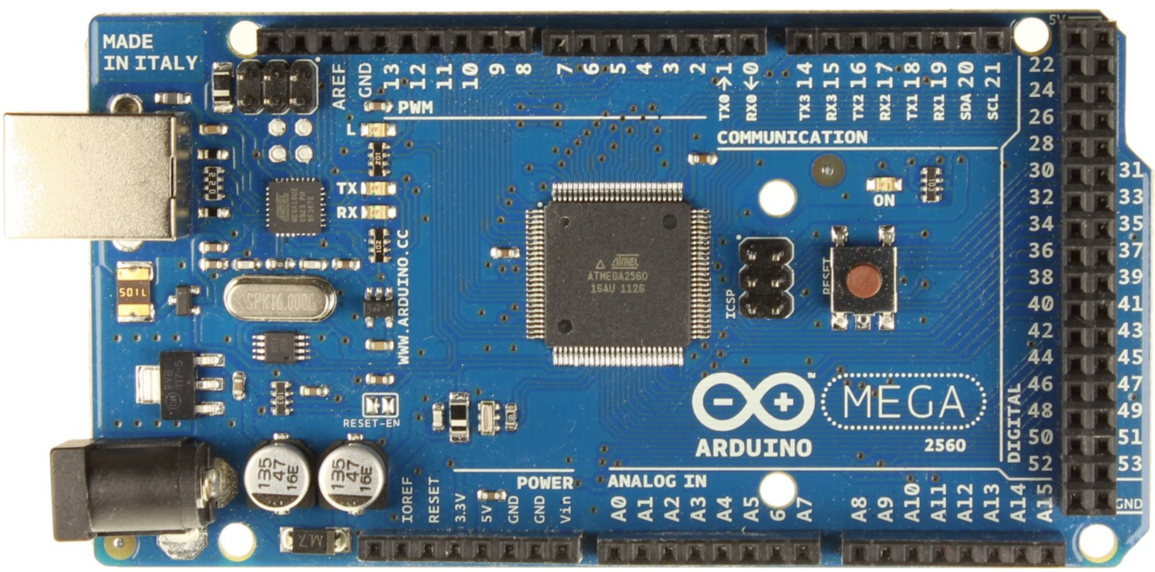
ii) Cross-platform: The Arduino software runs on Windows, Macintosh OSX, and Linux operating systems.

iii) Clear programming environment: The Arduino programming environment is easy-to-use for beginners, yet flexible enough for advanced users to take advantage of as well. For teachers, it's conveniently based on the Processing programming environment, so students learning to program in that environment will be familiar with the look and feel of Arduino

iv) Open source and extensible software: The Arduino software is published as an open source tool, available for extension by experienced programmers. The language can be expanded through C++ libraries, and people wanting to understand the technical details can make the leap from Arduino to the AVR-C programming language on which it's based.

ⅴ) Open source and extensible hardware: The Arduino is based on Atmel's ATMEGA8 and ATMEGA168 microcontrollers. The plans for the modules are published under a Creative Commons license, so experienced circuit designers can make their own version of the module, extending it and improving it. Even relatively inexperienced users can build the breadboard version of the module in order to understand how it works.

2. Arduino Mega 2560 (The entire section was taken from official Arduino website: See reference [1])



<Fig2: Arduino Mega 2560 Microcontroller board>

The Arduino Mega 2560 is a microcontroller board based on the ATmega2560 . It has 54 digital input/output pins (of which 15 can be used as PWM outputs), 16 analog inputs, 4 UARTs (hardware serial ports), a 16 MHz crystal oscillator, a USB connection, a power jack, an ICSP header, and a reset button. It contains everything needed to support the microcontroller; simply connect it to a computer with a USB cable or power it with a AC-to-DC adapter or battery to get started. The Mega is compatible with most shields designed for the Arduino Duemilanove or Diecimila [1].

A. Arduino Mega 2560 Specification

Microcontroller	ATmega2560
Operating Voltage	5V
Input Voltage (recommended)	7-12V
Input Voltage (limits)	6-20V
Digital I/O Pins	54 (of which 15 provide PWM output)
Analog Input Pins	16
DC Current per I/O Pin	40 mA
DC Current for 3.3V Pin	50 mA
Flash Memory	256 KB of which 8 KB used by bootloader
SRAM	8 KB
EEPROM	4 KB
Clock Speed	16 MHz

B. Power

The Arduino Mega can be powered via the USB connection or with an external power supply. The power source is selected automatically. External (non-USB) power can come either from an AC-to-DC adapter (wall-wart) or battery. The adapter can be connected by plugging a 2.1mm center-positive plug into the board's power jack. Leads from a battery can be inserted in the Gnd and Vin pin headers of the POWER connector.

The board can operate on an external supply of 6 to 20 volts. If supplied with less than 7V, however, the 5V pin may supply less than five volts and the board may be unstable. If using more than 12V, the voltage regulator may overheat and damage the board. The recommended range is 7 to 12 volts.

C. MEMORY

The ATmega2560 has 256 KB of flash memory for storing code (of which 8 KB is used for the bootloader), 8 KB of SRAM and 4 KB of EEPROM (which can be read and written with the EEPROM library).

D. COMMUNICATION

The Arduino Mega2560 has a number of facilities for communicating with a computer, another Arduino, or other microcontrollers. The ATmega2560 provides four hardware UARTs for TTL (5V) serial communication. An ATmega16U2 (ATmega 8U2 on the revision 1 and revision 2 boards) on the board channels one of these over USB and provides a virtual com port to software on the computer (Windows machines will need a .inf file, but OSX and Linux machines will recognize the board as a COM port automatically. The Arduino software includes a serial monitor which allows simple textual data to be sent to and from the board. The RX and TX LEDs on the board will flash when data is being transmitted via the ATmega8U2/ATmega16U2 chip and USB connection to the computer (but not for serial communication on pins 0 and 1).

A SoftwareSerial library allows for serial communication on any of the Mega2560's digital pins.The ATmega2560 also supports TWI and SPI communication. The Arduino software includes a Wire library to simplify use of the TWI bus; see the documentation for details. For SPI communication, use the SPI library.

E. Programming

The Arduino Mega can be programmed with the Arduino software. The ATmega2560 on the Arduino Mega comes preburned with a boot loader that allows you to upload new code to it without the use of an external hardware programmer. It communicates using the original STK500 protocol.

3. Hardware Architecture

Besides Arduino Mega2560 board, our PSD has following modules preinstalled in it:

A. Bluetooth

"Bluetooth allows you to easily connect mobile phones, notebook or desktop PCs, handheld devices, and printers over short distances (30 feet) without using a cable. Enabled devices send and receive information using radio signals. The technology was developed by the Bluetooth SIG(Special Interest Group) promoter and member companies, so mobile products could communicate without wires. Bluetooth capable products allow you to print images and documents from Laptop, Desktop or handheld devices, synchronize information between items and connect to other Bluetooth devices such as keyboards, mice, and headsets without cables" [3].

B. Global Positioning System (GPS)

"The Global Positioning System (GPS) is a space-based satellite navigation system that provides location and time information in all weather conditions, anywhere on or near the Earth where there is an unobstructed line of sight to four or more GPS satellites. GPS works in any weather conditions, anywhere in the world, 24 hours a day. GPS satellites circle the earth twice a day in a very precise orbit and transmit signal information to earth. GPS receivers take this information and use triangulation to calculate the user's exact location. Essentially, the GPS receiver compares the time a signal was transmitted by a satellite with the time it was received. The time difference tells the GPS receiver how far away the satellite is. Now, with distance measurements from a few more satellites, the receiver can determine the user's position and display it on the unit's electronic map. Today's GPS receivers are extremely accurate[4]”. This position can be tagged with the data from patient’s medical devices and make it more useful. For example, if the pulse oximeter shows low oxygen levels and patient is in an elevated location, the physician will likely ask the patient to go to lower altitude rather than diagnose him anemic.

C. INERTIAL MEASUREMENT UNIT (IMU)

“An inertial measurement unit (IMU) is an electronic device that measures and reports on a craft's velocity, orientation, and gravitational forces, using a combination of accelerometers and gyroscopes, sometimes also magnetometers. IMUs are typically used to maneuver aircraft. Recent developments allow for the production of IMU-enabled GPS devices. An IMU allows a GPS to work when GPS-signals are unavailable, such as in tunnels, inside buildings, or when electronic interference is present. A wireless IMU is known as a WIMU. The data collected from the IMU's sensors allows a computer to track a craft's position, using a method known as dead reckoning.” [5]

D. WI-FI

“Wi-Fi is a popular technology that allows an electronic device to exchange data or connect to the internet wirelessly using radio waves. The name is a contraction of "Wireless Fidelity", and was stated to be a play on the audiophile term Hi-Fi. Many devices can use Wi-Fi, e.g. personal computers, video-game consoles, smartphones, some digital cameras, tablet computers and digital audio players. These can connect to a network resource such as the Internet via a wireless network access point. Wi-Fi can be less secure than wired connections (such as Ethernet) because an intruder does not need a physical connection. Web pages that use SSL are secure but unencrypted internet access can easily be detected by intruders" [6]

E. Liquid Crystal Display (LCD)

"A liquid-crystal display(LCD) is a flat panel display, electronic visual display, or video display that uses the light modulating properties of liquid crystals. Liquid crystals do not emit light directly. LCDs are available to display arbitrary images(as in a general-purpose computer display) or fixed images which can be displayed or hidden, such as preset words, digits, and 7-segment displays as in a digital clock. They use the same basic technology, except that arbitrary images are made up of a large number of small pixels, while other displays have larger elements.

LCDs are used in a wide range of applications including computer monitors, televisions, instrument panels, aircraft cockpit displays, and signage. They are common in consumer devices such as video players, gaming devices, clocks, watches, calculators, and telephones, and have replaced cathode ray tube (CRT) displays in most applications. They are available in a wider range of screen sizes than CRT and plasma displays, and since they do not use phosphors, they do not suffer image burn-in. LCDs are, however susceptible to image persistence." [7]

4. The Advanced Encryption Standard (AES) (The entire section is a part of a chapter taught at Purdue. See reference [8])

A. SALIENT FEATURE OF AES

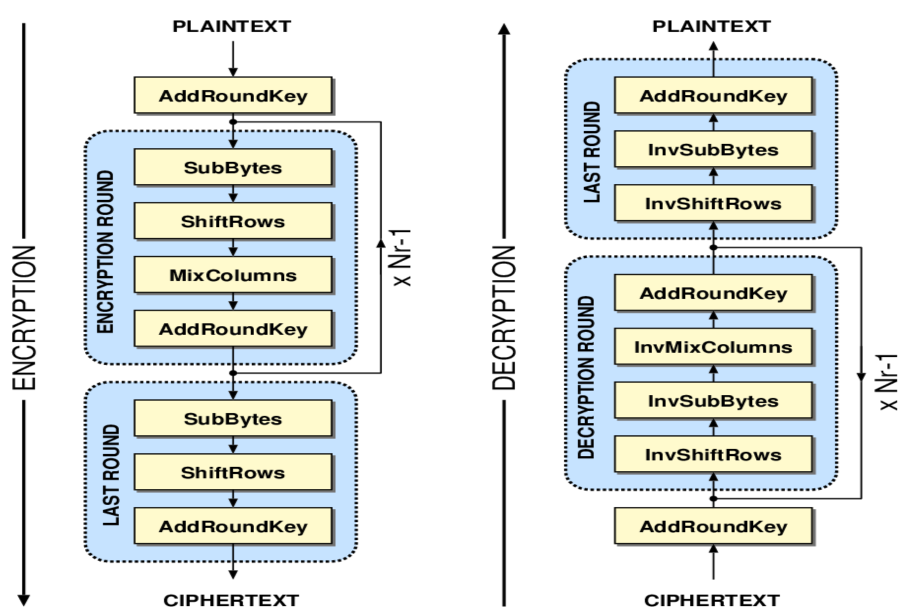
AES is a block cipher with a block length of 128 bits. It allows for three different key lengths: 128, 192, or 256 bits. AES Encryption consists of 10 rounds of processing for 128-bit keys, 12 rounds for 192-bit keys, and 14 rounds for 256-bit keys. Except for the last round in each case, all other rounds are identical. Each round of processing includes one single-byte based substitution step, a row-wise permutation step, a column-wise mixing step, and the addition of the round key. The order in which these four steps are executed is different for encryption and decryption. To appreciate the processing steps used in a single round, it is best to think of a 128-bit block as consisting of a 4×4 matrix of bytes, arranged as follows:

$$\begin{bmatrix} \text{byte0} & \text{byte4} & \text{byte8} & \text{byte12} \\ \text{byte1} & \text{byte5} & \text{byte9} & \text{byte13} \\ \text{byte2} & \text{byte6} & \text{byte10} & \text{byte14} \\ \text{byte3} & \text{byte7} & \text{byte11} & \text{byte15} \end{bmatrix}$$

Therefore, the first four bytes of a 128-bit input block occupy the first column in the 4 × 4 matrix of bytes. The next four bytes occupy the second column, and so on. The 4 × 4 matrix of bytes is referred to as the state array. Each round of processing works on the input state array and produces an output state array. The output state array produced by the last round is rearranged into a 128-bit output block. Unlike DES, the decryption algorithm differs substantially from the encryption algorithm. Although, overall, the same steps are used in encryption and decryption, the order in which the steps are carried out is different. Whereas AES requires the block size to be 128 bits, the original Rijndael cipher works with any block size (and any key size) that is a multiple of 32 as long as it exceeds 128. The state array for the different block sizes still has only four rows in the Rijndael cipher. However, the number of columns depends on size of the block. For example, when the block size is 192, the Rijndael cipher requires a state array to consist of 4 rows and 6 columns.

AES uses a substitution-permutation network in a more general sense. Each round of processing in AES involves byte-level substitutions followed by word-level permutations. The nature of substitutions and permutations in AES allows for a fast software implementation of the algorithm.

B. Working And Structure of AES



<Fig3: Encryption and Decryption Process>

Before any round-based processing for encryption can begin, the input state array is XORed with the first four words of the key schedule. The same thing happens during decryption — except that now we XOR the ciphertext state array with the last four words of the key schedule. For encryption, each round consists of the following four steps:

I. Substitute bytes
II. Shift rows
III. Mix columns
IV. Add round key.

The last step consists of XORing the output of the previous three steps with four words from the key schedule. For decryption, each round consists of the following four steps:

I. Inverse shift rows
II. Inverse substitute bytes
III. Add round key
IV. Inverse mix columns.

The third step consists of XORing the output of the previous two steps with four words from the key schedule. Note the differences between the order in which substitution and shifting operations are carried out in a decryption round vis-a-vis the order in which similar operations are carried out in an encryption round. The last round for encryption does not involve the “Mix columns” step. The last round for decryption does not involve the “Inverse mix columns” step [8].

5. Types of memory in an Arduino device

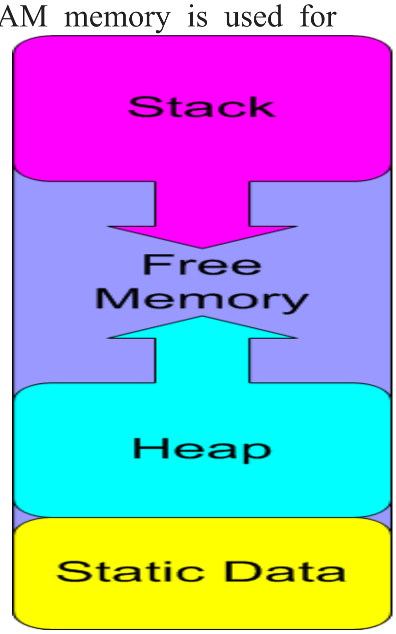
There are three types of memory in an Arduino:

A. Flash Memory

Flash memory is used to store program image and any initialized data. The flash is usually used to hold the executables (and perhaps other static data) for the device. It is possible to execute program code from flash, but one can't modify data in flash memory from your executing code. To modify the data, it must first be copied into SRAM. Flash memory has a finite lifetime of about 100,000 write cycles. So if 10 programs are uploaded 10 a day, every day for the next 27 years, one might wear it out.

B. SRAM

SRAM or **Static Random Access Memory**, can be read and written from executing program. This is where temporary variables are stored. SRAM memory is used for several purposes by a running program:



- **Static Data** - This is a block of reserved space in SRAM for all the global and static variables from program. For variables with initial values, the runtime system copies the initial value from Flash when the program starts.
- **Heap** - The heap is for dynamically allocated data items. The heap grows from the top of the static data area up as data items are allocated.

- **Stack** - The stack is for local variables and for maintaining a record of interrupts and function calls. The stack grows from the top of memory down towards the heap. Every interrupt, function call and/or local variable allocation causes the stack to grow. Returning from an interrupt or function call will reclaim all stack space used by that interrupt or function.

Most memory problems occur when the stack and the heap collide. When this happens, one or both of these memory areas will be corrupted with unpredictable results. In some cases it will cause an immediate crash. In others, the effects of the corruption may not be noticed until much later.

C. EEPROM

EEPROM is another form of non-volatile memory that can be read or written from executing program. It can only be read byte-by-byte, so it can be a little awkward to use. The EEPROM is used to store long-term information developed during the device's use. It is also slower than SRAM and has a finite lifetime of about 100,000 write cycles (you can read it as many times as you want). While it can't take the place of precious SRAM, there are times when it can be very useful!

IMPLEMENTATION OF AES IN PSD

1. Issues with implementing AES in Arduino

A major concern using Arduino as a PSD is the limited memory available. The difference between the Arduino microcontrollers and a general purpose computer is the sheer amount of memory available. The Arduino we are using has only 256K bytes of Flash memory, 8K bytes of SRAM and 4K bytes of EEPROM. That is 100,000 times LESS physical memory than a low-end PC! And that's not even counting the disk drive!

As described above, AES encryption requires 10 rounds of processing for 128-bit keys, 12 rounds for 192-bit keys, and 14 rounds for 256-bit keys. Each round of processing includes one single-byte based substitution step, a row-wise permutation step, a column-wise mixing step, and the addition of the round key. The process of encryption will require some memory space to store temporary results and the final encrypted results which go in flash. Key will be in the EEPROM. The data to be encrypted, any intermediate temporary results, and the encrypted block would probably go in SRAM

We need to determine if we can fit reasonable cryptographic software (probably AES) on this device, while still leaving room for other functionality. Alternatively, if we use AES crypto, how much of our space will be available for other operations.

2. Solution of the problems

Working in this minimalist environment, resources should be used wisely.

A. Installing AES on Arduino

First step towards solving the problem was to download the latest Arduino integrated development environment (IDE) software which is an open source and is available on Arduino's official website. Arduino IDE 1.0.5 is downloaded for this project. The next step was to install AES on Arduino. With some research, I was able to find an AES library that supports 128, 192 and 256 bit key sizes. This library can be found here: http://utter.chaos.org.uk:/~markt/AES-library.zip

This library was downloaded under Arduino ->library folder and imported to the Arduino IDE using sketch -> import library tabs.

B. Running codes to check the functionalities of AES Library

Code for Encryption and Decryption: We ran the code to produce the following output:

I. Plain text
II. Encrypted text (with varying block size)
III. Decrypted text (with varying block size)
IV. Encryption and decryption using 128, 192 and 256 bit key size
V. Time taken by each and every encryption and decryption process

The code for encryption and decryption is as follows:

```
#include <AES.h>

AES aes ;

byte key[] =
{
  0x80, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
} ;

byte plain[] =
{
  // 0xf3, 0x44, 0x81, 0xec, 0x3c, 0xc6, 0x27, 0xba, 0xcd, 0x5d, 0xc3, 0xfb, 0x08, 0xf2, 0x73, 0xe6
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0xC0, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0xE0, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0xF0, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
} ;

byte my_iv[] =
{
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01,
} ;

byte cipher [4*N_BLOCK] ;
byte check [4*N_BLOCK] ;

void loop ()
{}


void setup ()
{
  Serial.begin (57600) ;
  Serial.print ("testng mode") ;

  prekey_test () ;

//  otfly_test () ;
//  otfly_test256 () ;
}

void prekey (int bits, int blocks)
{
  byte iv [N_BLOCK] ;

  long t0 = micros () ;
  byte succ = aes.set_key (key, bits) ;
  long t1 = micros()-t0 ;
  Serial.print ("set_key ") ; Serial.print (bits) ; Serial.print (" ->") ; Serial.print ((int) succ) ;
  Serial.print (" took ") ; Serial.print (t1) ; Serial.println ("us") ;
  t0 = micros () ;
  if (blocks == 1)
    succ = aes.encrypt (plain, cipher) ;
  else
  {
    for (byte i = 0 ; i < 16 ; i++)
      iv[i] = my_iv[i] ;
    succ = aes.cbc_encrypt (plain, cipher, blocks, iv) ;
  }
  t1 = micros () - t0 ;
  Serial.print ("encrypt ") ; Serial.print ((int) succ) ;
  Serial.print (" took ") ; Serial.print (t1) ; Serial.println ("us") ;

  t0 = micros () ;
  if (blocks == 1)
    succ = aes.decrypt (cipher, plain) ;
  else
  {
    for (byte i = 0 ; i < 16 ; i++)
      iv[i] = my_iv[i] ;
    succ = aes.cbc_decrypt (cipher, check, blocks, iv) ;
  }
  t1 = micros () - t0 ;
  Serial.print ("decrypt ") ; Serial.print ((int) succ) ;
  Serial.print (" took ") ; Serial.print (t1) ; Serial.println ("us") ;

  for (byte ph = 0 ; ph < (blocks == 1 ? 3 : 4) ; ph++)
  {
    for (byte i = 0 ; i < (ph < 3 ? blocks*N_BLOCK : N_BLOCK) ; i++)
    {
      byte val = ph == 0 ? plain[i] : ph == 1 ? cipher[i] : ph == 2 ? check[i] : iv[i] ;
      Serial.print (val>>4, HEX) ; Serial.print (val&15, HEX) ; Serial.print (" ") ;
    }
    Serial.println () ;
  }
```

```
}

void prekey_test ()
{
  prekey (128, 4) ;
  prekey (192, 3) ;
  prekey (256, 2) ;
  prekey (128, 1) ;
  prekey (192, 1) ;
  prekey (256, 1) ;
}
```

<Fig3: Program for encrypting and decrypting data>

The following output was obtained from the code in Fig3 :

```
testng modeset_key 128 ->0 took 336us
encrypt 0 took 2276us
decrypt 0 took 2996us
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
93 E2 C5 24 3D 68 39 EA C5 85 03 91 91 92 F7 AE 59 E6 FF 16 BD 1F 1E F1 01 97 89 F4 C5 FF 6B 86 46 2B 74 60 73 F0 2B DE 23 46 03 09 DD D9 8D 30 51 06 A7 74 18 93 F2 56 CA C9 23 EA 71 5F E0 B1
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
51 06 A7 74 18 93 F2 56 CA C9 23 EA 71 5F E0 B1
set_key 192 ->0 took 376us
encrypt 0 took 2040us
decrypt 0 took 2692us
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
78 D6 7A 81 CE 94 C1 24 7D 25 C6 81 AB 73 BD 12 6E F7 AC 82 64 83 77 DB 44 FD 9D E9 4F B6 8E F4 D0 E3 EA E2 B9 ED F3 E3 37 08 DC 16 77 4E 66 19
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
D0 E3 EA E2 B9 ED F3 E3 37 08 DC 16 77 4E 66 19
set_key 256 ->0 took 464us
encrypt 0 took 1580us
decrypt 0 took 2092us
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
E1 4E 80 9C 6E 70 C9 CE 8D C0 D8 94 12 8E C7 83 84 58 F5 49 44 00 58 25 5A E2 28 56 CF DF 92 E6
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
84 58 F5 49 44 00 58 25 5A E2 28 56 CF DF 92 E6
set_key 128 ->0 took 336us
encrypt 0 took 540us
decrypt 0 took 708us
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0E DD 33 D3 C6 21 E5 46 45 5B D8 BA 14 18 BE C8
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
set_key 192 ->0 took 384us
encrypt 0 took 644us
decrypt 0 took 852us
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
DE 88 5D C8 7F 5A 92 59 40 82 D0 2C C1 E1 B4 2C
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
set_key 256 ->0 took 464us
encrypt 0 took 756us
decrypt 0 took 992us
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
E3 5A 6D CB 19 B2 01 A0 1E BC FA 8A A2 2B 57 59
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

<Fig4: Output showing encryption and decryption process and amount of time it took>

Checking test vectors: The output for code is showing the following scenarios:

I. Varying size key (128, 192 and 256)

II. Varying plain text and its cipher text

III. Monte Carlo

```
#include <AES.h>

AES aes ;

byte key [2*N_BLOCK] ;
byte plain [N_BLOCK] ;
byte iv [N_BLOCK] ;
byte cipher [N_BLOCK] ;
byte check [N_BLOCK] ;

void loop ()
{}


void setup ()
{
  Serial.begin (57600) ;
  Serial.print ("AES library test vectors") ;

  monte_carlo (128) ;

  for (int keysize = 128 ; keysize <= 256 ; keysize += 64)
    {
      prekey_test_var_plaintext (keysize) ;
      prekey_test_var_key (keysize) ;
    }
}


void prekey_test_var_plaintext (int bits)
{
  Serial.println () ;
  Serial.print ("ECB Varying Plaintext ") ;
  Serial.print (bits) ;
  Serial.println (" bits") ;
  Serial.println () ;

  byte succ ;
  set_bits (bits, key, 0) ;  // all zero key
  succ = aes.set_key (key, bits) ;
  if (succ != SUCCESS)
    Serial.println ("Failure set_key") ;
```

<Fig5: Test Vector Code Part-1>

```
    {
      Serial.print ("COUNT = ") ; Serial.println (bitcount-1) ;
      print_value ("KEY = ", key, bits) ;
      set_bits (128, plain, bitcount) ;

      print_value ("PLAINTEXT = ", plain, 128) ;

      succ = aes.encrypt (plain, cipher) ;
      if (succ != SUCCESS)
        Serial.println ("Failure encrypt") ;

      print_value ("CIPHERTEXT = ", cipher, 128) ;

      succ = aes.decrypt (cipher, check) ;
      if (succ != SUCCESS)
        Serial.println ("Failure decrypt") ;

      //print_value ("CHECK = ", check, 128) ;
      check_same (plain, check, 128) ;
      Serial.println () ;
    }
}


void prekey_test_var_key (int bits)
{
  Serial.println () ;
  Serial.print ("ECB Varying Key ") ;
  Serial.print (bits) ;
  Serial.println (" bits") ;
  Serial.println () ;

  byte succ ;
  set_bits (128, plain, 0) ;

  for (int bitcount = 1 ; bitcount <= bits ; bitcount++)
    {
      set_bits (bits, key, bitcount) ;  // all zero key
      succ = aes.set_key (key, bits) ;
      if (succ != SUCCESS)
        Serial.println ("Failure set_key") ;
      Serial.print ("COUNT = ") ; Serial.println (bitcount-1) ;
      print_value ("KEY = ", key, bits) ;
```

<Fig6: Test Vector Code Part-2>

```
      Serial.println ("Failure encrypt") ;

    print_value ("CIPHERTEXT = ", cipher, 128) ;

    succ = aes.decrypt (cipher, check) ;
    if (succ != SUCCESS)
      Serial.println ("Failure decrypt") ;

    check_same (plain, check, 128) ;
    Serial.println () ;
  }
}

void set_bits (int bits, byte * a, int count)
{
  bits >>= 3 ;
  byte bcount = count >> 3 ;
  for (byte i = 0 ; i < bcount ; i++)
    a [i] = 0xFF ;
  if ((count & 7) != 0)
    a [bcount++] = 0xFF & (0xFF00 >> (count & 7)) ;
  for (byte i = bcount ; i < bits ; i++)
    a [i] = 0x00 ;
}

void check_same (byte * a, byte * b, int bits)
{
  bits >>= 3 ;
  for (byte i = 0 ; i < bits ; i++)
    if (a[i] != b[i])
      {
        Serial.println ("Failure plain != check") ;
        return ;
      }
}

char * hex = "0123456789abcdef" ;


void print_value (char * str, byte * a, int bits)
{
  Serial.print (str) ;
  bits >>= 3 ;
```

<Fig7: Test Vector Code Part-3>

```
  for (int i = 0 ; i < bits ; i++)
    {
      byte b = a[i] ;
      Serial.print (hex [b >> 4]) ;
      Serial.print (hex [b & 15]) ;
    }
  Serial.println () ;
}

byte monteplain [] =
  { 0xb9, 0x14, 0x5a, 0x76, 0x8b, 0x7d, 0xc4, 0x89,
    0xa0, 0x96, 0xb5, 0x46, 0xf4, 0x3b, 0x23, 0x1f } ;
byte montekey []   =
  { 0x13, 0x9a, 0x35, 0x42, 0x2f, 0x1d, 0x61, 0xde,
    0x3c, 0x91, 0x78, 0x7f, 0xe0, 0x50, 0x7a, 0xfd } ;

void monte_carlo (int bits)
{
  Serial.println () ;
  Serial.print ("Monte Carlo ") ;
  Serial.print (bits) ;
  Serial.println (" bits") ;
  Serial.println () ;
  byte succ ;
  for (int i = 0 ; i < 16 ; i++)
  {
    plain [i] = monteplain [i] ;
    key [i] = montekey [i] ;
  }
  for (int i = 0 ; i < 100 ; i++)
    {
      Serial.print ("COUNT = ") ; Serial.println (i) ;
      print_value ("KEY = ", key, bits) ;
      print_value ("PLAINTEXT = ", plain, 128) ;
      succ = aes.set_key (key, bits) ;
      for (int j = 0 ; j < 1000 ; j++)
        {
          succ = aes.encrypt (plain, cipher) ;
          aes.copy_n_bytes (plain, cipher, 16) ;
        }
      print_value ("CIPHERTEXT = ", cipher, 128) ;
      Serial.println () ;
```

<Fig8: Test Vector Code Part-4>

```
      Serial.println () ;
      if (bits == 128)
        {
          for (byte k = 0 ; k < 16 ; k++)
            key [k] ^= cipher [k] ;
        }
      else if (bits == 192)
        {
        }
      else
        {
        }
    }
}
```

<Fig9: Test Vector Code Part-5>

Part of output of the test vectors looks this this:

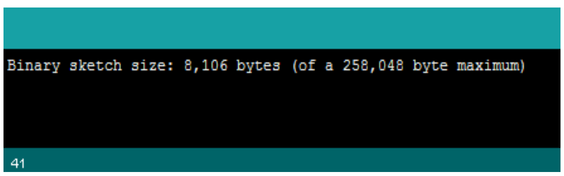
```
Monte Carlo 128 bits

COUNT = 0
KEY = 139a35422f1d61de3c91787fe0507afd
PLAINTEXT = b9145a768b7dc489a096b546f43b231f
CIPHERTEXT = d7c3ffac9031238650901e157364c386

COUNT = 1
KEY = c459caeebf2c42586c01666a9334b97b
PLAINTEXT = d7c3ffac9031238650901e157364c386
CIPHERTEXT = bc3637da2daf8fcf7c68bb28c143a0a4

COUNT = 2
KEY = 786ffd349283cd971069dd42527719df
PLAINTEXT = bc3637da2daf8fcf7c68bb28c143a0a4
CIPHERTEXT = 9c88a8db798f48df1ac4936afa959eac

COUNT = 3
KEY = e4e755efeb0c85480aad4e28a8e28773
PLAINTEXT = 9c88a8db798f48df1ac4936afa959eac
CIPHERTEXT = b87aaa1c76a775d94c2ddf82abe5c66e

COUNT = 4
KEY = 5c9dfff39dabf091468091aa0307411d
PLAINTEXT = b87aaa1c76a775d94c2ddf82abe5c66e
CIPHERTEXT = 79ee212734f14d1bf5a59d46e8c2fa34

COUNT = 5
KEY = 2573ded4a95abd8ab3250cecebc5bb29
PLAINTEXT = 79ee212734f14d1bf5a59d46e8c2fa34
CIPHERTEXT = 09df49135aeb8e373a19fa457ab280a0

COUNT = 6
KEY = 2cac97c7f3b133bd893cf6a991773b89
PLAINTEXT = 09df49135aeb8e373a19fa457ab280a0
CIPHERTEXT = c52263efa6379209d17e87ac250615cb

COUNT = 7
KEY = e98ef4285586a1b458427105b4712e42
PLAINTEXT = c52263efa6379209d17e87ac250615cb
CIPHERTEXT = 336bed017e10a247ee92989862431163
```

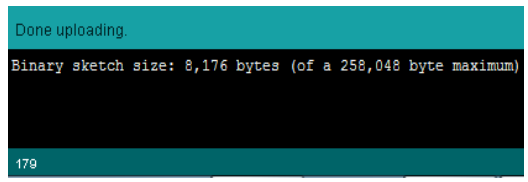
<Fig10: Test Vector code output>

C. CHECKING THE AMOUNT OF MEMORY USED AND REMAINING

I. Flash

The amount of flash memory used can be found out at the bottom part of the sketch. One can see the amount of bytes being used after uploading the program. Both encryption & decryption and test vector code was taking approximately 8000 bytes out of total 258,048 bytes in Arduino.





<Fig11:Flash memory used by encryption> <Fig12: Flash memory used by Test Vector>

II. EEPROM

EEPROM usage is in fully controlled by user, i.e. we have to read and write each byte to a specific address. So if we want to save 128 bit key in the EEPROM, it will take only 16 bytes in EEPROM. Similarly, a 192 and 256 key will take 24 and 32 bytes respectively.

III. SRAM

SRAM usage is more dynamic and therefore more difficult to measure. The free_ram() function below is one way to do this. You can add this function definition to your code, then call it from various places in your code to report the amount of free SRAM.

```
void setup ()
{
  Serial.begin (57600) ;
  Serial.print ("testng mode") ;

  prekey_test () ;
 Serial.print ("Memory free: ") ;  Serial.println(freeRam());

//  otfly_test () ;
//  otfly_test256 () ;
}
```

<Fig13:Calling free_ram() function in encryption/decryption code>

```
int freeRam () {
  extern int __heap_start, *__brkval;
  int v;
  return (int) &v - (__brkval == 0 ? (int) &__heap_start : (int) __brkval);
}
```

<Fig14: free_ram() function in encryption/decryption code>

The function free_ram() actually reports the space between the heap and the stack but it does not report any de-allocated memory that is buried in the heap. Buried heap space is not usable by the stack, and may be fragmented enough that it is not usable for many heap allocations either. The space between the heap and the stack is what we really need to monitor if we are trying to avoid stack crashes.

```
set_key 192 ->0 took 384us
encrypt 0 took 648us
decrypt 0 took 852us
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
DE 88 5D C8 7F 5A 92 59 40 82 D0 2C C1 E1 B4 2C
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
set_key 256 ->0 took 464us
encrypt 0 took 756us
decrypt 0 took 992us
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
E3 5A 6D CB 19 B2 01 A0 1E BC FA 8A A2 2B 57 59
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
SRAM free: 6921
```

<Fig15:Output showing free SRAM memory>

Summary of space available and space used:

Type of Memories	Total space available	Space used	Space Remaining
Flash	256 KB	8.79 KB	247.2 KB
EEPROM	4 KB	0.015 KB	3.98 KB
SRAM	8 KB	1.4 KB	6.8 KB

<Table1:Amount of flash, EEPROM and SRAM available/used/remaining>

3. Achievements of the project

AES was successfully installed in Arduino and was tested against many of the test-vectors (key varying, plaintext varying, Monte Carlo). It was able to perform encryption and decryption with varying key sizes!

This project was successful in attaining main goal i.e. to determine

- whether AES can be installed in the Arduino platform: A simple AES encryption decryption process was installed.
- the amount of memory available for other device functionalities: Using separate techniques for observing the amount of memory used, we were able to find out the exact amount of memory that will be used/saved.

In addition to performing encryption and decryption, this project made sure that the time taken for the encryption/decryption is considerably low!! Time that encryption took was calculated at each and every step to make sure that we don't face timing issues in future!! Some speed/timing information is:

128 bit, key setup 0.37ms

128 bit, ECB, encryption 0.58ms / block (27.5kB/s)

128 bit, ECB, decryption 0.77ms / block (20.5kB/s)

192 bit, key setup 0.41ms

192 bit, ECB, encryption 0.71ms / block (22.5kB/s)

192 bit, ECB, decryption 0.92ms / block (17.5kB/s)

256 bit, key setup 0.52ms

256 bit, ECB, encryption 0.82ms / block (19.5kB/s)

256 bit, ECB, decryption 1.09ms / block (14.5kB/s)

CONCLUSION

This project determined that AES can be implemented in Arduino and there will be sufficient space for other functionalities too. A quick observation of the table above shows that AES implementation took very less space and most of the space available in Arduino is unused and can be used for other functionalities.

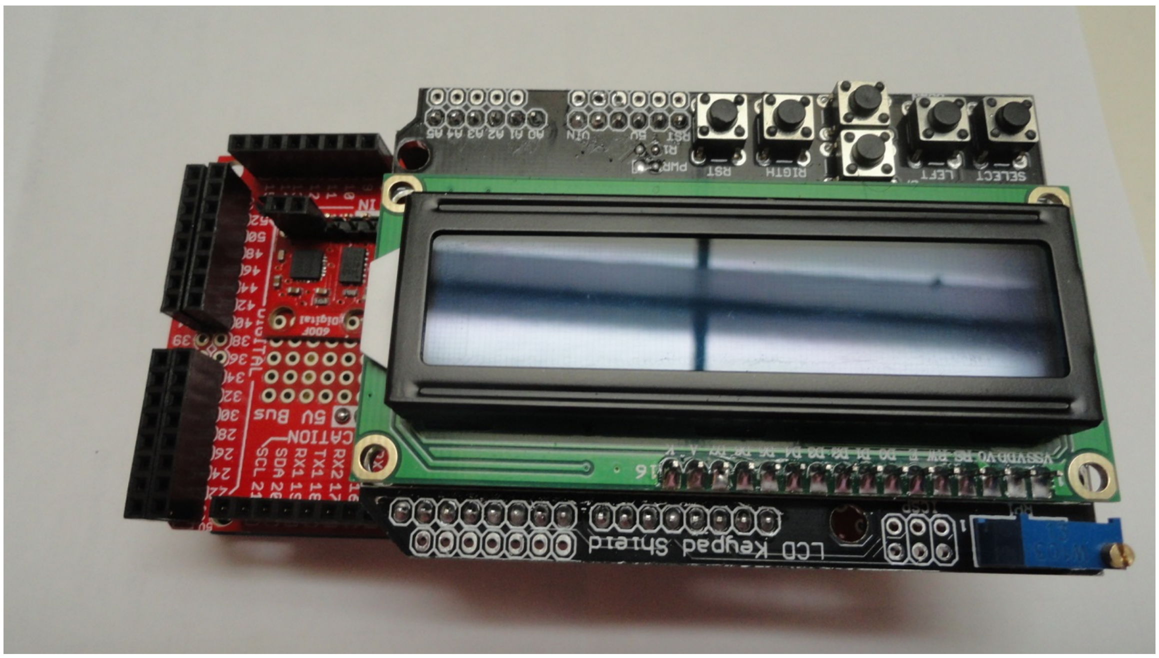
While dealing with time issues was not a part of this project, this project made sure that the time is kept in close watch. It was observed that the amount of time taken by encryption/decryption process is less and so ensures that implementing security wouldn't affect the efficiency of the device time wise.

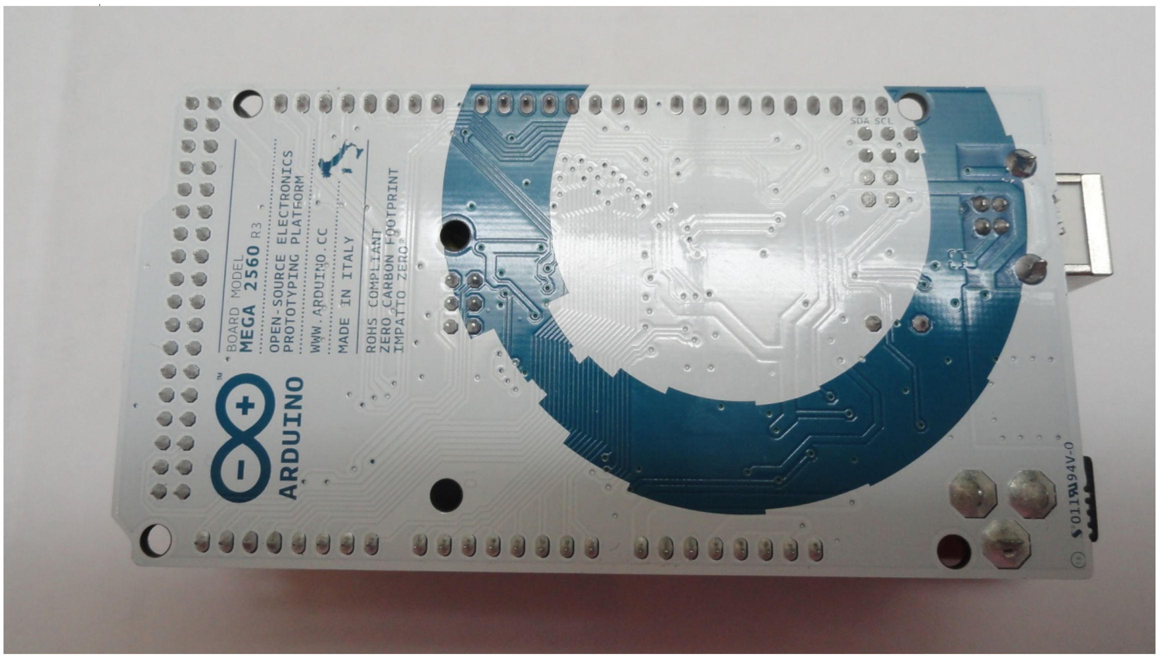
REFERENCES

1. Arduino Official Website, http://arduino.cc/
2. Symantec Official Blog, http://www.symantec.com/connect/blogs/webinar-anatomy-breach-medical-devices-under-cyber-attack
3. http://h10025.www1.hp.com/ewfrf/wc/bluetoothDocument?lc=en&dlc=en&cc=us&docname=c00222918&printable=yes&encodeUrl=true&
4. http://en.wikipedia.org/wiki/Global_Positioning_System
5. http://en.wikipedia.org/wiki/Inertial_measurement_unit
6. http://en.wikipedia.org/wiki/Wi-Fi
7. http://en.wikipedia.org/wiki/Liquid-crystal_display
8. https://engineering.purdue.edu/kak/compsec/NewLectures/Lecture8.pdf
9. Vahab Pournaghshband, Majid Sarrafzadeh, Peter Reiher, "Securing Legacy Mobile Medical Devices", *Wireless Mobile Communication and Healthcare, Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, volume 61, pp 163-172, 2013.

APPENDIX

Pictures of Arduino with all modules:





ARDUINO
BOARD MODEL
MEGA 2560 R3
OPEN-SOURCE ELECTRONICS
PROTOTYPING PLATFORM
WWW.ARDUINO.CC
MADE IN ITALY
ROHS COMPLIANT
ZERO CARBON FOOTPRINT
IMPATTO ZERO
SDA SCL