Report 2

Alex Hu, Shawn Hsiao, Shih-Hao Yeh

ABSTRACT:

In this project, we start to build up our first embedded product which is a medical monitoring system. By building up the system, we learned to work with formal design methodologies such as UML and block diagram, build up simple operating system with task queue and share data between task. Also, we utilized the knowledge learned from the previous project to build communication between two devices and show information on a TFT screen.

INTRODUCTION:

Medical monitoring systems are must for people in the world nowadays, especially for people like elders and patients. The basic use case will be in the hospital (Fig. 1). These systems' basic requirements are to collect data and able to transmit data. Also, some may also need the devices to be cheap, light weight and so on. Arduino is such a device that can fulfill all the requirements (even though it might not be sufficient for some tasks that require fast communication and accurate measurement, it is good enough for our demo purpose) and that is why we are using it as our hardware.

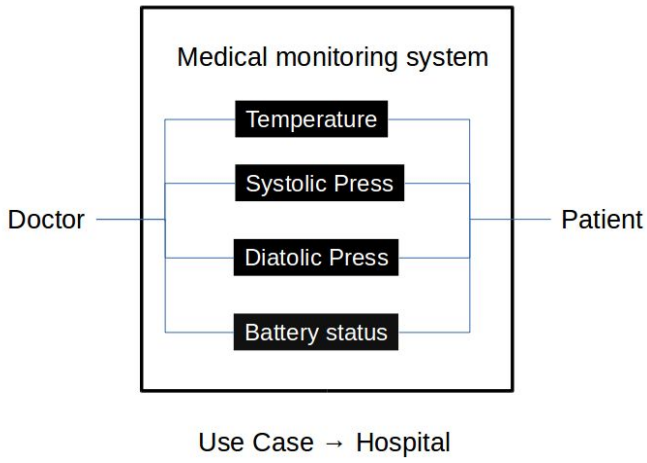

Fig 1 Use case

In our case, we were in the first stage of building up the medical monitoring system. Therefore, the first step was to use formal design methodologies such as UML to set up our device's requirement and architecture. We divided the system into two part. The first part will be the system control subsystem and the other part will be the peripheral subsystem. The system control subsystem will maintain the task queue and looping all the time, while the peripheral subsystem

will collect data from measurement devices (we use simulated data for now) and transmit the data to the system control subsystem.

In the system control subsystem, we use Arduino ATMega board as the hardware. We define 5 tasks, including measurement, compute, display, status and warning. Each task will have different data type and different response. On the other hand, we use Arduino Uno board as the peripheral subsystem that serve as the measurement device. It will return 4 kinds of measurements, including temperature, systolic press, diastolic press, and pulse.

The working flow is that the system control subsystem will first send request to the peripheral subsystem and return back with the measurement data. Then the data will be compute to other format and display on a TFT screen. If the data is out of the range we defined, the display will show warning to the user. The procedure will keep looping until we stop it.

DESIGN SPECIFICATION:

- temperatureRaw: Increment by 2 every even numbered time and decrement by 1 every odd numbered time until the value of the variable exceeds 50.
- systolicPressRaw: Increment by 3 every even numbered time and decremented by 1 every odd numbered time until the value of the variable exceeds 100. When the diastolic measurement is complete, repeat the process.
- diastolicPressRaw: Decrement the variable by 2 every even numbered time and incremented by 1 every odd numbered time until the value of the variable drops below 40. When the systolic measurement is complete, repeat the process.
- pulseRateRaw: Decrement the variable by 1 every even numbered time and increment by 3 every odd numbered time until the value of the variable exceeds 40.
- batteryStatus: Decrease by 1 until the value is equal to 0, and stop decreasing.
- Blood pressure is shown red when greater than 120 or less than 80.
- Temperature is shown red when less than 36.1 or greater than 37.8.
- Pulse rate us shown red when less than 60 or greater than 100.
- Task queue is executed every 5 seconds.
- Each task is represented as a task control block written in a data struct
- Data associated with each block is written as a data struct pointing to the required variables
- Major computation done on ATMega board with Uno board doing measurement.
- ATMega sends request for measurement to Uno and Uno sends back data to ATMega through serial communication.

SOFTWARE IMPLEMENTATION:

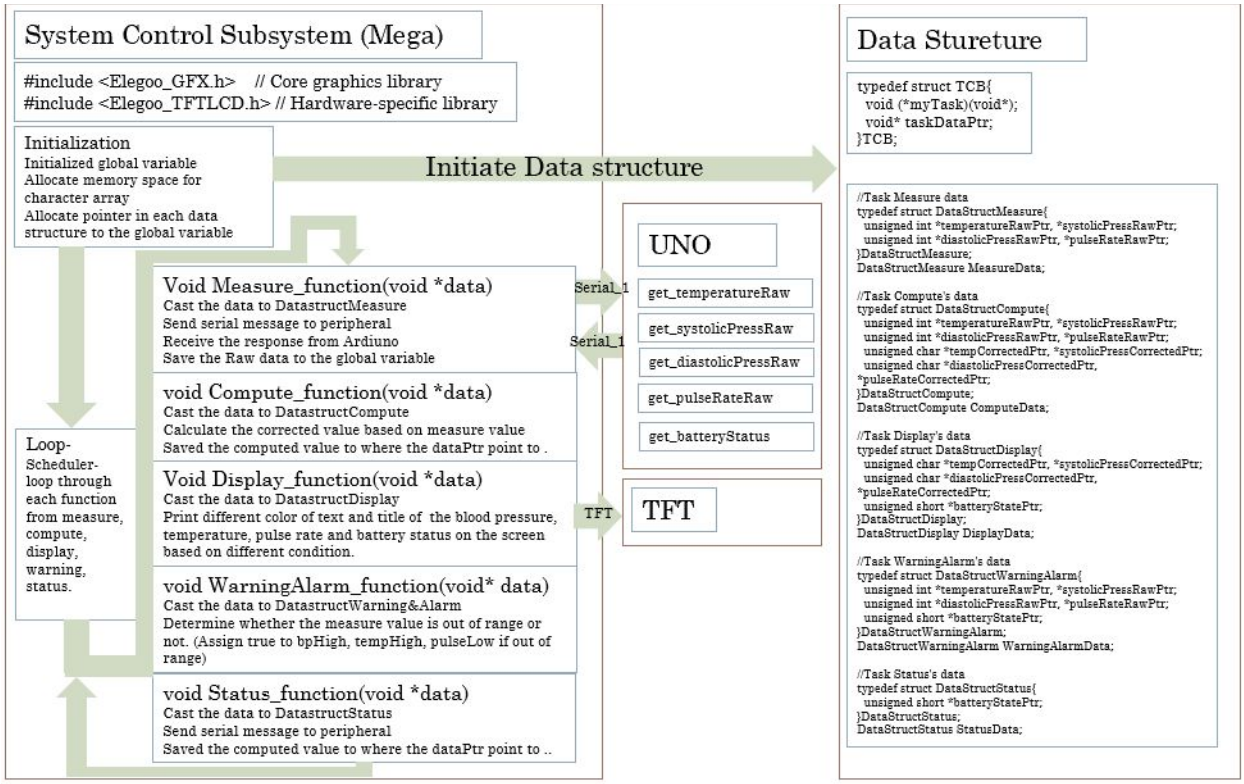

Figure 2 Control Diagram

The control diagram displays the relationship between each function in both system control subsystem as well as in the peripheral subsystem. The first step includes initialization of all related data structure such as TCB for each function (refer to Appendix A for detail), initial value and data struct pointing to these value. After initialization, the scheduler in the loop section will start to call each function (Measure_function, Compute_function...etc) in sequence of 5 second. And each function has its related response, for example connecting with peripheral device arduino or TFT. More description about each function will be provided in following section, but more details are listed in appendix A.

System Control Subsystem (Mega)

Initialization:

- Include the necessary libraries to display on TFT: include <Elegoo_GFX.h> as core graphics library and include <Elegoo_TFTLCD.h> as hardware-specific library.
- Initialize parameters for TFT display: define read/write pins and color codes, set the background to black.
- Declare and initialize global variables for temperature, blood pressure, pulse rate, battery status, warning and alarm.

- Define and initialize data structure for each task's data: assign address of global variables the task would be using to the pointers in the data structure.
- Define and initialize data structure for task control block: assign address of function and data structure to the pointers in the TCB struct.

```
/*Define tasks*/
#define numTask 5
typedef struct TCB{
  void (*myTask)(void*);
  void* taskDataPtr;
}TCB;
TCB Measure, Compute, Display;
TCB WarningAlarm, Status;
TCB* taskque[numTask];
```

- Define and initialize pin to toggle after executing the whole task queue.
- Initialize serial communication baud rate.

Loop:

- Scheduler
  5 tasks are scheduled sequentially from the task queue in the order of: warning -> display -> measure -> compute-> status. The task queue is executed every 5 second by using the delay function.
- Measure_function
  The measure_function firstly cast the input void data pointer to DatastructMeasure data type. Then it send serial message to peripheral (UNO), and wait for it's response. Once get the string reply which encoded with 4 number. This function parse through the string and split it into four integer value (represent temperature, sys pressure, dis pressure, pulse rate). Finally, it save them into global variable where the data pointer point to.

- Compute_function
  The compute_function cast the input void data pointer to DatastructCompute data type and calculate the corrected value based on the raw value stored in global variable which can be retrieved by casted data pointer. Once the calculation is done, it save the value inside the corrected value through data pointer.

- Display_function
  The display_function cast the input void data pointer to DatastructDisplay data type and print different color of text and title of the blood pressure, temperature, pulse rate and battery status on the screen based on different condition which are defined in warning and Alarm function.

- WarningAlarm_function

The warningAlarm_function casts the data to DatastructWarning&Alarm and determine whether the measure value is out of range or not. If yes, assign true to global variable bpHigh, tempHigh, pulseLow.

- Status_funcion
  The Status_funcion cast the input void data pointer to DatastructStatus data type. Then it send serial message to peripheral (UNO), and wait for it's response. Once get the string reply which encoded with 1 number. This function convert the string into integer value and save it into global variable where the data pointer point to.

Peripheral Subsystem (UNO):
In this part, instead of using real measurement from devices, we set up simple simulation data for our demo purpose. The peripheral subsystem will wait for the system call from the system control subsystem. There are two kinds of system call that will come from the system control subsystem. One is the measurement call, the other is the status system call. We use "1" and "2" to distinguish the request call from the control subsystem. When measurement call, Peripheral subsystem will return measure data from 4 different measurement, which are temperature, systolic press, diastolic pressure and pulse rate. Since we need to send back the 4 data at a time, we concatenate the four data using string and add a space between each measurement. It's easy for the control subsystem to decode and get the value for the 4 measurement. When status call, we send back the battery status.

- get_temperatureRaw

  This function's requirement is increment the variable by 2 every even numbered time the function is called and decrement by 1 every odd numbered time the function is called until the value of the variable exceeds 50. In order to count the called time, a naive way of doing this is to setup a global variable integer, and add one every time it is called. However, given that our system will be running forever until we stop it, using this method can simply explode the maximum of integer. And the same thing will happen even if we use long or long long integer. Therefore, we come up with the idea to use positive one and negative one as the indicator of the even and odd time it's called. First, the counter was set to 1. Every time it is called, we multiply the counter by -1. This make the counter turning back and forth from 1 to -1. We then use this feature to write different response function for the measurement.

- get_systolicPressRaw

This function is related to the get_diastolicPressRaw. They have constrain on each other. Increment the variable by 3 every even numbered time the function is called and decremented by 1 every odd numbered time the function is called until the value of the variable exceeds 100. Also, when the diastolic measurement is complete, repeat the process. This is trying to simulate the process of measuring the blood pressure. Basically, when systolic pressure is increasing, the diastolic will hold, and vice versa. However, since the initial value for the two variable are set to the middle point, the two function will go on simultaneously when the system start.

To achieve the function described above, we set up the same counter as in the get_temperatureRaw(). On the other hand, we add global boolean variables indicating the completeness of the function for get_systolicPressRaw() and get_diastolicPressRaw(). The variable can be see from the other function. In this function, when get_diastolicPressRaw() is completed, the value of systolic press will be reset and start to increase again.

- get_diastolicPressRaw
  This function is implemented with almost the same concept in get_systolicPressRaw() but in the opposite way and the requirement is decrement the variable by 2 every even numbered time the function is called and incremented by 1 every odd numbered time the function is called until the value of the variable drops below 40.

- get_pulseRateRaw
  This function is implemented with almost the same concept in get_temperatureRaw(). The requirement is decrement the variable by 1 every even numbered time the function is called and increment by 3 every odd numbered time the function is called until the value of the variable exceeds 40. And then reverse the process.

- get_batteryStatus
  This function is much easier than the previous 4 function. The requirement is only decrement by 1 every time it's called. Therefore, our implementation is just decrease by 1 until the value is equal to 0, and we stop decreasing.

TEST PLAN:

We need to check that the measured values and computed values are correct as specified. In addition, the values out of range should be displayed as red color. Last, the task queue should be executed within a certain time.

TEST SPECIFICATION:

- temperatureRaw: Increment by 2 every even numbered time and decrement by 1 every odd numbered time until the value of the variable exceeds 50.
- systolicPressRaw: Increment by 3 every even numbered time and decremented by 1 every odd numbered time until the value of the variable exceeds 100. When the diastolic measurement is complete, repeat the process.
- diastolicPressRaw: Decrement the variable by 2 every even numbered time and incremented by 1 every odd numbered time until the value of the variable drops below 40. When the systolic measurement is complete, repeat the process.
- pulseRateRaw: Decrement the variable by 1 every even numbered time and increment by 3 every odd numbered time until the value of the variable exceeds 40.
- batteryStatus: Decrease by 1 until the value is equal to 0, and stop decreasing.
- Blood pressure is shown red when greater than 120 or less than 80.
- Temperature is shown red when less than 36.1 or greater than 37.8.
- Pulse rate us shown red when less than 60 or greater than 100.
- Task queue is executed every 5 seconds.

TEST CASES:

- Show the raw data in the serial plotter for verification.
- Show the corrected data on the TFT display.
- Observe the pin output to see the execution time of the task queue.

RESULT ANALYSIS:

- Raw data Plot for verification- all line in both figure 3 and figure 4 matches the design specification described above. (These figures are generated by UNO Serial plot)

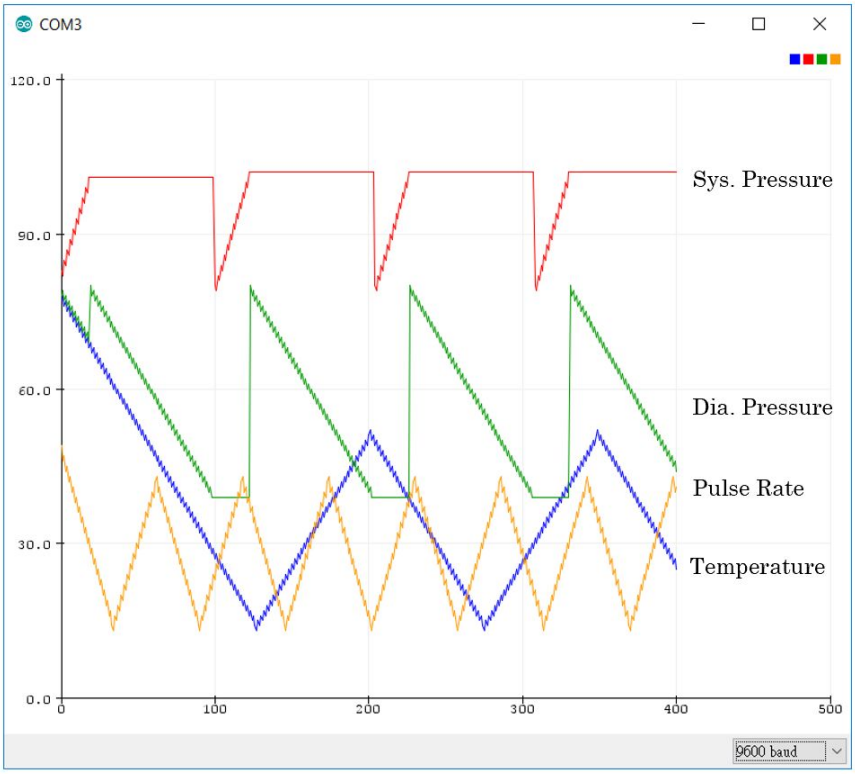

Figure 3 Value from UNO get_Measure function

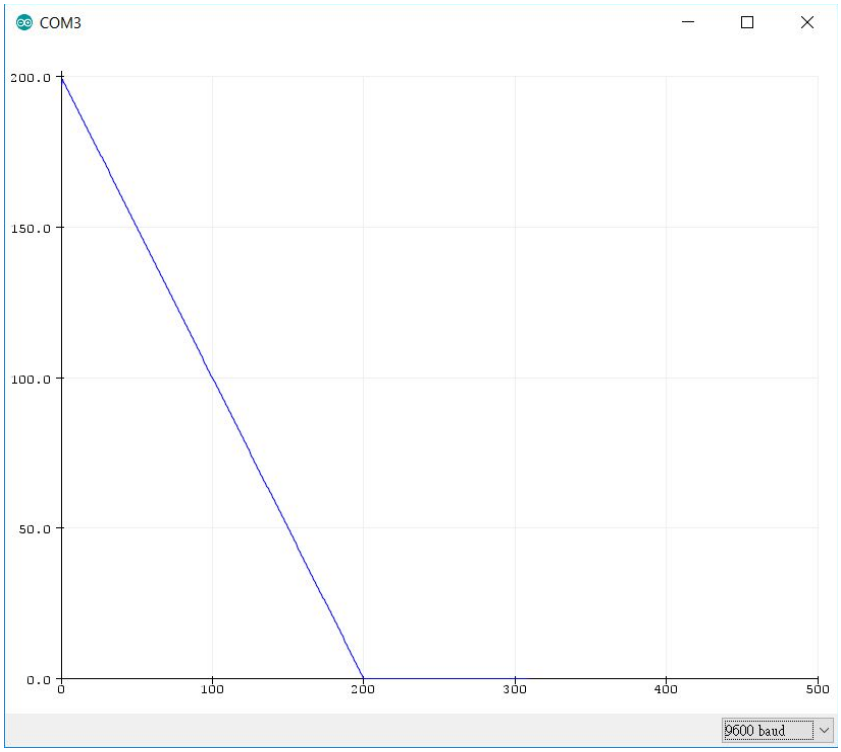

Figure 4 Value from UNO get_Satus function

- Show the corrected data on the TFT display.

This part will be discuss in the error analysis given to the initial value is explicitly designed by us. Therefore, to check the calculation and color of text is much easier than using value during running. But, the picture of TFT is shown below.

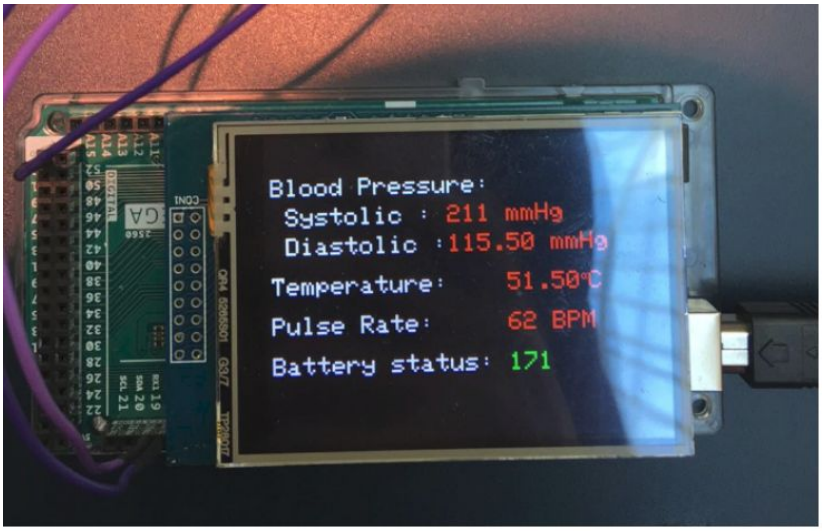

Figure 5 Result of TFT

- Observe the pin output to see the execution time of the task queue.
  We list the execution time of each function in the below table.

Table 1 Execution time

Measure	1017 ms
Compute	0 ms
Display	752 ms
Warning & Alarm	0 ms
Status	1008 ms

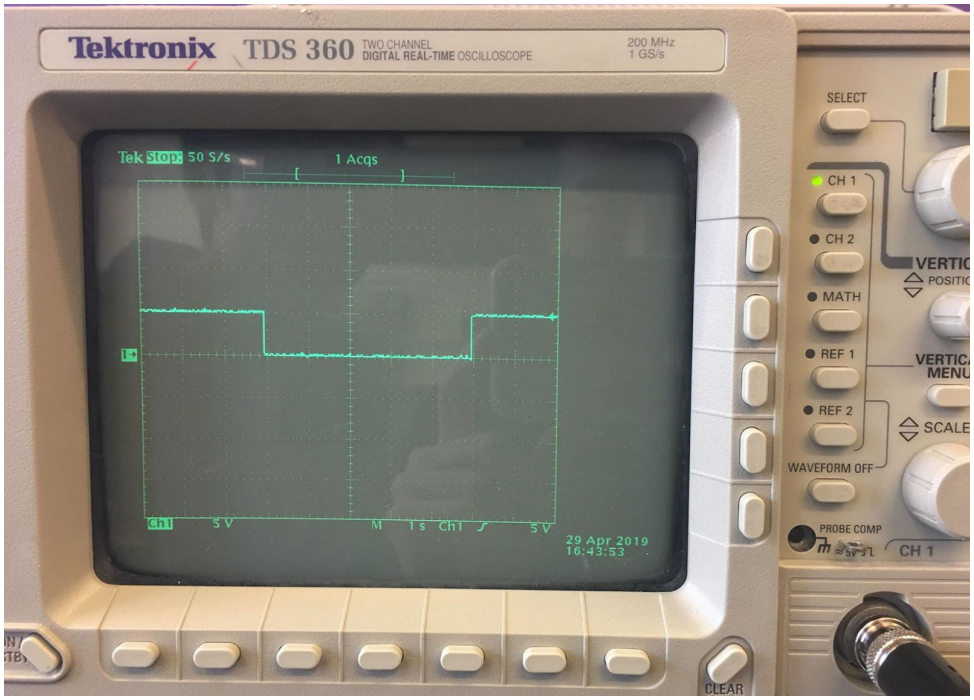

Figure 6 Result of oscilloscope

Our result from oscilloscope match with our design which is 5 second.

ERROR ANALYSIS:

1.In order to see the initial value, we rearrange the order of the task from Measure, Compute, Display, Warning and Status to Warning, Display, Measure, Compute, and Status. The new order allow the initial value be displayed on the screen before measure and compute function change the initial value. We can view our change by checking the value shown on the screen. (ex. Battery =200) . For verification of the correctness of color and computation in the compute function, the initial value for temperature is 75 degree C, so after compute with formula: 5+0.75*(*(data->temperatureRawPtr), the corrected value is 61.25 which accord with what shown on the TFT display. (Also the color is red because temperature is out of range).

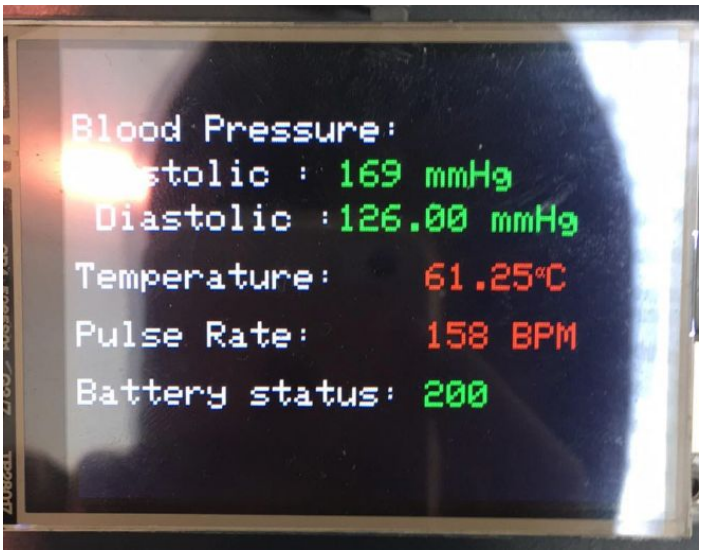

Figure 6 Initial Value (Corrected)

2.Queue Time measurement problem

In order to compensate the delay time to 5 second, we record the initial time of function. However, we record the wrong start time (the start time of last function in the queue), so the duration of entire queue becomes 7 second (Debug from Oscilloscope). After adjust the record time to the first function, the duration of queue become 5 second.

3.Battery Status

Realistically, once the battery status decrease to zero, the entire device should turn off. But our system remain zero for battery status. To simulate the device, the screen should turn black if battery is zero.

SUMMARY:

In this project, we learned to use struct that contain void pointer and pointer to function to adapt to different tasks. This can make the task initialization process become more efficient and elegant. Also this allow us to use an array of the TCB struct to looping through different tasks with different return type. On the other hand, by using UNO as the peripheral subsystem, we again practiced the serial communication between two devices. Last but not least, we utilized TFT screen to display the data to the user and update the color of the text when the data is out of range.

CONCLUSION:

This project is a simplify case of the medical monitoring system. Many of the feature we built now may not satisfied the requirement in the real application, for example, the measurement and

warning task should be running without interrupt all the time instead of putting them in a loop. This might need some hardware interrupt feature or some threading feature. However, this is a good practice for us to start with a big product and get familiar with the whole design process.

CONTRIBUTION:

Alex:

- TCB define and initialize.
- Data struct for each TCB define and initialize.
- Schedule the task queue.
- List design spec and testing procedures.

Shawn:

- Develop all the function of measure, compute, display, warning, and status and explain their function in the software implementation part.
- Draw the control diagram in software implementation.
- Explain the result analysis and error analysis .

Shih-Hao:

- Implement the peripheral subsystem including 4 measurement function and battery status.
- Serial communication between ATMega and UNO.
- Elaborate abstract, introduction, summary and conclusion in report.

APPENDIX:

System Control Subsystem (Mega) Code

```
#include <Elegoo_GFX.h>    // Core graphics library
#include <Elegoo_TFTLCD.h> // Hardware-specific library

// The control pins for the LCD can be assigned to any digital or
// analog pins...but we'll use the analog pins as this allows us to
// double up the pins with the touch screen (see the TFT paint example).
#define LCD_CS A3 // Chip Select goes to Analog 3
#define LCD_CD A2 // Command/Data goes to Analog 2
#define LCD_WR A1 // LCD Write goes to Analog 1
#define LCD_RD A0 // LCD Read goes to Analog 0

#define LCD_RESET A4 // Can alternately just connect to Arduino's reset pin

// When using the BREAKOUT BOARD only, use these 8 data lines to the LCD:
```

```
// For the Arduino Uno, Duemilanove, Diecimila, etc.:
//   D0 connects to digital pin 8  (Notice these are
//   D1 connects to digital pin 9   NOT in order!)
//   D2 connects to digital pin 2
//   D3 connects to digital pin 3
//   D4 connects to digital pin 4
//   D5 connects to digital pin 5
//   D6 connects to digital pin 6
//   D7 connects to digital pin 7
// For the Arduino Mega, use digital pins 22 through 29
// (on the 2-row header at the end of the board).

// Assign human-readable names to some common 16-bit color values:
#define  BLACK   0x0000
#define BLUE    0x001F
#define RED     0xF800
#define GREEN   0x07E0
#define CYAN    0x07FF
#define MAGENTA 0xF81F
#define YELLOW  0xFFE0
#define WHITE   0xFFFF

Elegoo_TFTLCD tft(LCD_CS, LCD_CD, LCD_WR, LCD_RD, LCD_RESET);
// If using the shield, all control and data lines are fixed, and
// a simpler declaration can optionally be used:
// Elegoo_TFTLCD tft;


/*Shared varibles*/
//Measurements
unsigned int temperatureRaw=75, systolicPressRaw=80;
unsigned int diastolicPressRaw=80, pulseRateRaw=50;
//Display
unsigned char *tempCorrected=NULL, *systolicPressCorrected=NULL;
unsigned char *diastolicPressCorrected=NULL, *pulseRateCorrected=NULL;
//Status
unsigned short batteryState=200;
//Alarms
unsigned char bpOutOfRange=0,tempOutOfRange=0, pulseOutOfRange=0;
```

```
//Warning
bool bpHigh=false, tempHigh=false, pulseLow=false;


/*Define tasks*/
#define numTask 5
typedef struct TCB{
  void (*myTask)(void*);
  void* taskDataPtr;
}TCB;
TCB Measure, Compute, Display;
TCB WarningAlarm, Status;
TCB* taskque[numTask];

/*Define data structures*/
//Task Measure's data
typedef struct DataStructMeasure{
  unsigned int *temperatureRawPtr, *systolicPressRawPtr;
  unsigned int *diastolicPressRawPtr, *pulseRateRawPtr;
}DataStructMeasure;
DataStructMeasure MeasureData;

//Task Compute's data
typedef struct DataStructCompute{
  unsigned int *temperatureRawPtr, *systolicPressRawPtr;
  unsigned int *diastolicPressRawPtr, *pulseRateRawPtr;
  unsigned char *tempCorrectedPtr, *systolicPressCorrectedPtr;
  unsigned char *diastolicPressCorrectedPtr, *pulseRateCorrectedPtr;
}DataStructCompute;
DataStructCompute ComputeData;

//Task Display's data
typedef struct DataStructDisplay{
  unsigned char *tempCorrectedPtr, *systolicPressCorrectedPtr;
  unsigned char *diastolicPressCorrectedPtr, *pulseRateCorrectedPtr;
  unsigned short *batteryStatePtr;
}DataStructDisplay;
DataStructDisplay DisplayData;
```

```
//Task WarningAlarm's data
typedef struct DataStructWarningAlarm{
  unsigned int *temperatureRawPtr, *systolicPressRawPtr;
  unsigned int *diastolicPressRawPtr, *pulseRateRawPtr;
  unsigned short *batteryStatePtr;
}DataStructWarningAlarm;
DataStructWarningAlarm WarningAlarmData;

//Task Status's data
typedef struct DataStructStatus{
  unsigned short *batteryStatePtr;
}DataStructStatus;
DataStructStatus StatusData;

/*Tasks's function*/
void Measure_function(void *uncast_data){
  DataStructMeasure* data;
  data = (DataStructMeasure*)uncast_data;
  String  serialResponse;
  Serial1.write("1");
  int value[4];

  while ( !Serial1.available()){}
  serialResponse = Serial1.readStringUntil('\n');

  // Convert from String Object to String.
  char buf[serialResponse.length()+1];
  serialResponse.toCharArray(buf, sizeof(buf));
  char *p = buf;
  char *str;
  int i = 0;
  String cast_str;
  while ((str = strtok_r(p, " ", &p)) != NULL) // delimiter is the semicolon
  {
    cast_str=(String)str;
    value[i] = cast_str.toInt();
    i++;
  }
  //for(int j=0; j<4; j++) {
```

```
    //Serial.print("This is Measure value");
    //Serial.println(value[j]);
  //};

 *(data->temperatureRawPtr)=value[0];
  *(data->systolicPressRawPtr)=value[1];
  *(data->diastolicPressRawPtr)=value[2];
  *(data->pulseRateRawPtr)=value[3];
}

void Compute_function(void *uncast_data){
  DataStructCompute* data;
  data = (DataStructCompute *)uncast_data;
  sprintf((char*)data->tempCorrectedPtr, "%s",
String(5+0.75*(*(data->temperatureRawPtr))).c_str());
  sprintf((char*)data->systolicPressCorrectedPtr, "%s",
String(9+2*(*(data->systolicPressRawPtr))).c_str());
  sprintf((char*)data->diastolicPressCorrectedPtr, "%s",
String(6+1.5*(*(data->diastolicPressRawPtr))).c_str());
  sprintf((char*)data->pulseRateCorrectedPtr, "%s",
String(8+3*(*(data->pulseRateRawPtr))).c_str());
}


void Display_function(void *uncast_data){
  DataStructDisplay* data;
  data=(DataStructDisplay*)uncast_data;
   tft.setRotation(1);
   tft.setCursor(0, 30);
   tft.setTextSize(2);
   tft.setTextColor(WHITE);
   tft.println("Blood Pressure: ");
   tft.setTextSize(1);
   tft.println("        ");
   tft.setTextSize(2);
   tft.print(" Systolic : ");
   if (bpHigh==true){
      tft.setTextColor(RED,BLACK);
```

```
  tft.print((char*)(data->systolicPressCorrectedPtr));

  tft.println(" mmHg   ");}
else{
  tft.setTextColor(GREEN,BLACK);
  tft.print((char*)(data->systolicPressCorrectedPtr));
  tft.println(" mmHg   ");};

tft.setTextColor(WHITE);
tft.setTextSize(1);
tft.println("        ");
tft.setTextSize(2);
tft.print(" Diastolic :");
if (pulseLow==true){
  tft.setTextColor(RED,BLACK);

  tft.print((char*)(data->diastolicPressCorrectedPtr));
  tft.println(" mmHg ");}
else{
  tft.setTextColor(GREEN,BLACK);
  tft.print((char*)(data->diastolicPressCorrectedPtr));
  tft.println(" mmHg  ");};

tft.setTextColor(WHITE);
tft.println("        ");
tft.print("Temperature:    ");
if (tempHigh==true){tft.setTextColor(RED,BLACK);}
else{tft.setTextColor(GREEN,BLACK);};
tft.print((char*)(data->tempCorrectedPtr));
tft.setTextSize(1);
tft.print((char)223);
tft.setTextSize(2);
tft.println("C  ");


tft.setTextColor(WHITE);
tft.println("        ");
tft.print("Pulse Rate:     ");
if (pulseLow==true){tft.setTextColor(RED,BLACK);}
```

```
  else{tft.setTextColor(GREEN,BLACK);};
  tft.print((char*)(data->pulseRateCorrectedPtr));
  tft.println(" BPM ");


  tft.setTextColor(WHITE);
  tft.println("        ");
  tft.print("Battery status: ");
  if (batteryState<=20){tft.setTextColor(RED,BLACK);}
  else{tft.setTextColor(GREEN,BLACK);};
  tft.print(*(data->batteryStatePtr));
  tft.println("    ");

  tft.println();
  tft.println();
}

void WarningAlarm_function(void *uncast_data){
 DataStructWarningAlarm * data;
 data = ( DataStructWarningAlarm *)uncast_data;
 if(*(data->temperatureRawPtr)<36.1 || *(data->temperatureRawPtr)>37.8){
  tempOutOfRange=1;
    tempHigh=true;
  }else{
    tempHigh=false;
    tempOutOfRange=0;

 }
 if (*(data->systolicPressRawPtr)>120 || *(data->diastolicPressRawPtr)<80){
    bpOutOfRange=1;
    bpHigh=true;
  }else{
    bpHigh=false;
    bpOutOfRange=0;
 }

 if (*(data->pulseRateRawPtr)<60 || *(data->pulseRateRawPtr)>100){
    pulseOutOfRange=1;
    pulseLow=true;
```

```
   }else{
     pulseLow=false;
     pulseOutOfRange=0;

  }
}



void Status_function(void *uncast_data){
  String  serialResponse;
  DataStructStatus* data;
  int value;
  data=(DataStructStatus *)uncast_data;
  Serial1.write("2");

  while( !Serial1.available()) {}
  serialResponse = Serial1.readStringUntil('\n');

  // Convert from String Object to String.
  char buf[serialResponse.length()+1];
  serialResponse.toCharArray(buf, sizeof(buf));
  char *p = buf;
  char *str;

  int i = 0;
  String cast_str;
  while ((str = strtok_r(p, " ", &p)) != NULL) // delimiter is the semicolon
  {
   cast_str=(String)str;
   value = cast_str.toInt();
  }
  //for(int j=0; j<1; j++) {

  //  Serial.print("This is Status value");
    //Serial.println(value);
    //}

  *(data->batteryStatePtr)=value;
```

```
}

String taskName[numTask] = {"Measure", "Compute", "Display", "WarningAlarm", "Status"};
String message; //message of task time
#define taskqueFinishPin 30 //pin to be toggled after one cycle of task que
unsigned long qStart_time; //the start time of taskque
unsigned long start_time; //the start time of each task
unsigned long taskTime[numTask]; //store the execution time of each task

void setup() {

  //Allocate memory space for character array
  tempCorrected=(unsigned char*)malloc(sizeof(unsigned char)*10);
  systolicPressCorrected=(unsigned char*)malloc(sizeof(unsigned char)*10);
  diastolicPressCorrected=(unsigned char*)malloc(sizeof(unsigned char)*10);
  pulseRateCorrected=(unsigned char*)malloc(sizeof(unsigned char)*10);

  //Initialized task Measure
  Measure.myTask = Measure_function;
  MeasureData.temperatureRawPtr = &temperatureRaw;
  MeasureData.systolicPressRawPtr = &systolicPressRaw;
  MeasureData.diastolicPressRawPtr = &diastolicPressRaw;
  MeasureData.pulseRateRawPtr = &pulseRateRaw;
  Measure.taskDataPtr = &MeasureData;
  taskque[2] = &Measure;

  //Initialized task Compute
  Compute.myTask = Compute_function;
  ComputeData.temperatureRawPtr = &temperatureRaw;
  ComputeData.systolicPressRawPtr = &systolicPressRaw;
  ComputeData.diastolicPressRawPtr = &diastolicPressRaw;
  ComputeData.pulseRateRawPtr = &pulseRateRaw;
  ComputeData.tempCorrectedPtr = tempCorrected;
  ComputeData.systolicPressCorrectedPtr = systolicPressCorrected;
  ComputeData.diastolicPressCorrectedPtr = diastolicPressCorrected;
  ComputeData.pulseRateCorrectedPtr = pulseRateCorrected;
  Compute.taskDataPtr = &ComputeData;
  taskque[3] = &Compute;
```

```
//Initialized task Display
Display.myTask = Display_function;
DisplayData.tempCorrectedPtr = tempCorrected;
DisplayData.systolicPressCorrectedPtr = systolicPressCorrected;
DisplayData.diastolicPressCorrectedPtr = diastolicPressCorrected;
DisplayData.pulseRateCorrectedPtr = pulseRateCorrected;
DisplayData.batteryStatePtr = &batteryState;
Display.taskDataPtr = &DisplayData;
taskque[1] = &Display;

//Initialized task WarningAlarm
WarningAlarm.myTask = WarningAlarm_function;
WarningAlarmData.temperatureRawPtr = &temperatureRaw;
WarningAlarmData.systolicPressRawPtr = &systolicPressRaw;
WarningAlarmData.diastolicPressRawPtr = &diastolicPressRaw;
WarningAlarmData.pulseRateRawPtr = &pulseRateRaw;
WarningAlarmData.batteryStatePtr = &batteryState;
WarningAlarm.taskDataPtr = &WarningAlarmData;
taskque[0] = &WarningAlarm;

//Initialized task Status
Status.myTask = Status_function;
StatusData.batteryStatePtr = &batteryState;
Status.taskDataPtr = &StatusData;
taskque[4] = &Status;

//Initialized taskqueFinishPin
pinMode(taskqueFinishPin, OUTPUT);
digitalWrite(taskqueFinishPin, LOW);

//Initialized serial port 0 & 1
Serial.begin(9600);
Serial1.begin(9600);

//Initialized for TFT
Serial.println(F("TFT LCD test"));
#ifdef USE_Elegoo_SHIELD_PINOUT
Serial.println(F("Using Elegoo 2.4\" TFT Arduino Shield Pinout"));
#else
```

```
Serial.println(F("Using Elegoo 2.4\" TFT Breakout Board Pinout"));
#endif
Serial.print("TFT size is "); Serial.print(tft.width()); Serial.print("x"); Serial.println(tft.height());
tft.reset();
 uint16_t identifier = tft.readID();
 if(identifier == 0x9325) {
  Serial.println(F("Found ILI9325 LCD driver"));
} else if(identifier == 0x9328) {
  Serial.println(F("Found ILI9328 LCD driver"));
} else if(identifier == 0x4535) {
  Serial.println(F("Found LGDP4535 LCD driver"));
}else if(identifier == 0x7575) {
  Serial.println(F("Found HX8347G LCD driver"));
} else if(identifier == 0x9341) {
  Serial.println(F("Found ILI9341 LCD driver"));
} else if(identifier == 0x8357) {
  Serial.println(F("Found HX8357D LCD driver"));
} else if(identifier==0x0101)
{
    identifier=0x9341;
     Serial.println(F("Found 0x9341 LCD driver"));
}
else if(identifier==0x1111)
{
    identifier=0x9328;
     Serial.println(F("Found 0x9328 LCD driver"));
}
else {
  Serial.print(F("Unknown LCD driver chip: "));
  Serial.println(identifier, HEX);
  Serial.println(F("If using the Elegoo 2.8\" TFT Arduino shield, the line:"));
  Serial.println(F("  #define USE_Elegoo_SHIELD_PINOUT"));
  Serial.println(F("should appear in the library header (Elegoo_TFT.h)."));
  Serial.println(F("If using the breakout board, it should NOT be #defined!"));
  Serial.println(F("Also if using the breakout, double-check that all wiring"));
  Serial.println(F("matches the tutorial."));
  identifier=0x9328;

}
```

```
  tft.begin(identifier);
  tft.fillScreen(BLACK);
}

/*Scheduler*/
void loop() {
  qStart_time = millis();
  for (int i=0; i<numTask; i++){
    start_time = millis();
    (taskque[i]->myTask)(taskque[i]->taskDataPtr); //execute task
    taskTime[i] = millis() - start_time;
  }
  while(millis() - qStart_time < 5000){} //schedule whole taskque for 5 sec
  //toggle pin after one cycle of task que
  digitalWrite(taskqueFinishPin, !digitalRead(taskqueFinishPin));
  //show execution time for each task in serial monitor
  message = "";
  for (int i=0; i<numTask; i++){
    message += taskName[i] + ": " + taskTime[i] + " ms\n";
  }
  Serial.write(message.c_str());
}
```

Peripheral (UNO)

```
// initial value for the measurements
unsigned int temperatureRaw = 75;
unsigned int systolicPressRaw = 80;
unsigned int diastolicPressRaw = 80;
unsigned int pulseRateRaw = 50;
unsigned short batteryStatus = 200;

// count the execution time in each measurement. 1 means even time, -1 means odd time
int temperatureRaw_count = 1;
int systolicPressRaw_count = 1;
int diastolicPressRaw_count = 1;
int pulseRateRaw_count = 1;

// determine whether the task is complete
```

```
bool temperatureRaw_flip = true;
bool systolicPressRaw_flip = true; // if set to false, it means complete
bool systolic_reset = false;
bool diastolicPressRaw_flip = true; // if set to false, it means complete
bool diastolic_reset = false;
bool pulseRateRaw_flip = true;

// string for serial communication
String whichTask;
String measureData = "";

// set up serial port
void setup(){
  Serial.begin(9600);
}

// main loop
void loop(){
  if ( Serial.available() > 0 ) {
    whichTask = Serial.readStringUntil('\0');

    // if receive 1, return measurement
    if( whichTask == "1" ){
        measureData = "";
        measureData += String(get_temperatureRaw());
        measureData += " ";
        measureData += String(get_systolicPressRaw());
        measureData += " ";
        measureData += String(get_diastolicPressRaw());
        measureData += " ";
        measureData += String(get_pulseRateRaw());
        Serial.println(measureData);
    }

    // if receive 2, return battery status
    else if( whichTask == "2" ){
        measureData = String(get_batteryStatus());
        Serial.println(measureData);
    }
  }
}

unsigned int get_temperatureRaw(){
```

```
  // before temperatureRaw exceed 50
  if( temperatureRaw_flip ){
    if( temperatureRaw_count > 0 ) temperatureRaw += 2;
    else temperatureRaw -= 1;

    if( temperatureRaw > 50 ) temperatureRaw_flip = false;
  }
  // before temperatureRaw lower than 15
  else{
    if( temperatureRaw_count > 0 ) temperatureRaw -= 2;
    else temperatureRaw += 1;

    if( temperatureRaw < 15 ) temperatureRaw_flip = true;
  }

  // flip counter
  temperatureRaw_count *= -1 ;
  return temperatureRaw;
}

unsigned int get_systolicPressRaw(){

  // if diastolic is completed and haven't reset
  if( !diastolicPressRaw_flip && !systolic_reset ){
    systolicPressRaw = 80;
    systolic_reset = true;
  }

  // when first go out of range
  else if( systolicPressRaw > 100 && systolicPressRaw_flip == true ){
    systolicPressRaw_flip = false; // complete
    diastolicPressRaw_flip = true;
    systolic_reset = false;
  }

  // when out of range
  else if( systolicPressRaw > 100 && systolicPressRaw_flip == false ){
    systolicPressRaw = systolicPressRaw;
  }

  // when in the range
  else{
    if( systolicPressRaw_count > 0 ) systolicPressRaw += 3;
```

```
    else systolicPressRaw -= 1;
  }

  // flip counter
  systolicPressRaw_count *= -1 ;
  return systolicPressRaw;
}

unsigned int get_diastolicPressRaw(){

  // if systolic is completed and haven't reset
  if( !systolicPressRaw_flip && !diastolic_reset ){
    diastolicPressRaw = 80;
    diastolic_reset = true;
  }

  // when first go out of range
  else if( diastolicPressRaw < 40 && diastolicPressRaw_flip == true ){
    diastolicPressRaw_flip = false; // complete
    systolicPressRaw_flip = true;
    diastolic_reset = false;
  }

  // when go out of range
  else if( diastolicPressRaw < 40 && diastolicPressRaw_flip == false ){
    diastolicPressRaw = diastolicPressRaw;
  }

  // when in the range
  else{
    if( diastolicPressRaw_count > 0 ) diastolicPressRaw -= 2;
    else diastolicPressRaw += 1;
  }

  // flip counter
  diastolicPressRaw_count *= -1 ;
  return diastolicPressRaw;
}

unsigned int get_pulseRateRaw(){

  // before pulseRateRaw exceed 50
  if( pulseRateRaw_flip ){
```

```
    if( pulseRateRaw_count > 0 ) pulseRateRaw -= 1;
    else pulseRateRaw += 3;

    if( pulseRateRaw > 40 ) pulseRateRaw_flip = false;
   }

   // before pulseRateRaw lower than 15
   else{
    if( pulseRateRaw_count > 0 ) pulseRateRaw += 1;
    else pulseRateRaw -= 3;

    if( pulseRateRaw < 15 ) pulseRateRaw_flip = true;
   }

   // flip counter
   pulseRateRaw_count *= -1 ;
   return pulseRateRaw;
}

unsigned short get_batteryStatus(){
  if( batteryStatus > 0 ) batteryStatus -= 1;
  return batteryStatus;
}
```