Report 3

Alex Hu, Shawn Hsiao, Shih-Hao Yeh

ABSTRACT:
In this project, we implemented a more adaptive operating system by using double link list instead of using an array to store the TCB. Each TCB can be added to the queue dynamically. In the measurement part, we added a buffer array to store up to 8 historical data for each measure task. The warning and display task is further implemented with an alarm function, and will show notification when specific data is out of a specific range. When this happen, a communication task will also print the measurement data to serial monitor. Last but not least, we implemented an user interface that can allow user to choose which task to measurement.

INTRODUCTION:

When using a medical monitor device, it is common that we only want to measurement some specific task. In the previous project, we set all the tasks in an array and looping through all the task each time. This has one advantage, which is the easy design process. However, it is lack of flexibility. If we only want a specific measurement, we still need to wait for all the measurements to complete. Therefore, the idea of using double link list as the task queue comes up. By using double link list, we can easily add in and delete some task from the queue. By remembering the head pointer, we can run from the head pointer to the tail pointer and then use a scheduler to rearrange the task queue.

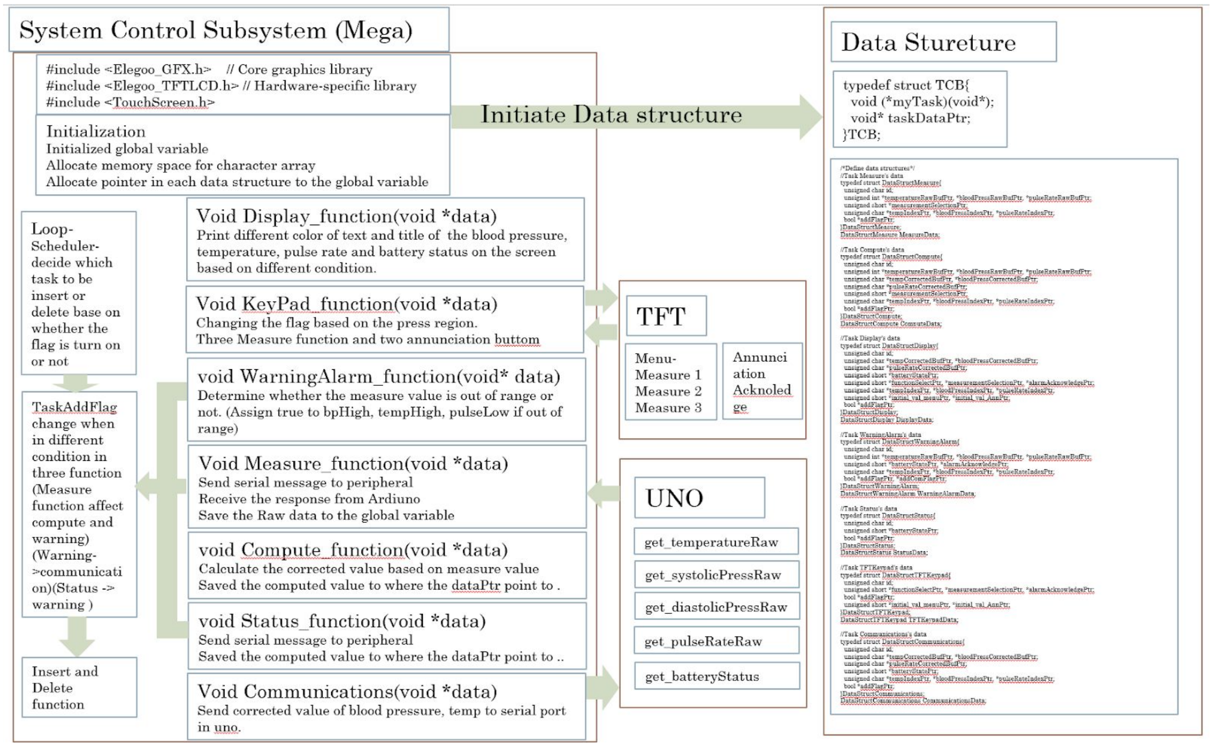
This time, we have 7 tasks, measurement, compute, warning, communication, display, status, and keypad task. Communication and keypad tasks are new from previous project. There are several changes in the previous defined tasks. In measurement and compute tasks, instead of using only one variable to store the current value, we use arrays to store the historical data. Most of the measurement tasks are still using simulation, but the pulse rate are using output interrupt from a function generator.

The keypad task need to take using input to setup different flag, changing the display mode from menu to annunciation. In the display task we also add in a function that is related to warning task. When the systolic blood pressure is out of specific range, the display text will turn into red, and wait for user to press a button called acknowledge. If the measurement is out of range again for 5 times, the alarm will show up again. This function is very useful in the real application of a monitor system. Last but not least, we also add a communication task to the task queue. This task will output the measurement data to a remote device (in this project we use serial monitor as demonstration). The task is functioning when warning occurs.

DESIGN SPECIFICATION:

- temperatureRawBuf, systolicPressRawBuf, diastolicPressRawBuf: Same simulation as last time but data is put in buffers.
- pulseRateRawBuf: Read in 0V to 3.3V square wave from function generator and use the external interrupt to count the number of pulses. Calculate the rate and put in the buffer.
- batteryStatus: Same as last time.
- A user interface with two pages on TFT
    - Menu: Selection for measurement
    - Annunciation: Show measurement values and acknowledgement button
- Blood pressure is shown orange when greater than 120 or less than 80, red when greater than upper bound 20%.
- Temperature is shown orange when less than 36.1 or greater than 37.8.
- Pulse rate is shown orange when less than 60 or greater than 100.
- A warning message is sent to the serial monitor when any warnings or alarms occur.
- Measurement is executed every 5 seconds by a dynamic scheduler.
- Each task is represented as a task control block written in a data struct with previous and next pointer.
- Data associated with each block is written as a data struct pointing to the required variables
- Dynamic scheduler uses double linked list to schedule the task queue.
- Major computation done on ATMega board with Uno board doing measurement.
- ATMega sends request for individual measurement to Uno and Uno sends back data to ATMega through serial communication.

SOFTWARE IMPLEMENTATION:



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

Initialization
Initialized global variable
Allocate memory space for character array
Allocate pointer in each data structure to the global variable

Loop-Scheduler-decide which task to be insert or delete base on whether the flag is turn on or not

TaskAddFlag change when in different condition in three function (Measure function affect compute and warning) (Warning->communication)(Status -> warning)

Insert and Delete function

Void Display_function(void *data)
Print different color of text and title of the blood pressure, temperature, pulse rate and battery status on the screen based on different condition.

Void KeyPad_function(void *data)
Changing the flag based on the press region.
Three Measure function and two annunciation buttom

void WarningAlarm_function(void* data)
Determine whether the measure value is out of range or not. (Assign true to bpHigh, tempHigh, pulseLow if out of range)

Void Measure_function(void *data)
Send serial message to peripheral
Receive the response from Ardiuno
Save the Raw data to the global variable

void Compute_function(void *data)
Calculate the corrected value based on measure value
Saved the computed value to where the dataPtr point to .

void Status_function(void *data)
Send serial message to peripheral
Saved the computed value to where the dataPtr point to ..

Void Communications(void *data)
Send corrected value of blood pressure, temp to serial port in uno.

```
typedef struct TCB{
  void (*myTask)(void*);
  void* taskDataPtr;
}TCB;
```

```
/*Define data structures*/
//Task Measure's data
typedef struct DataStructMeasure{
  unsigned char id;
  unsigned int *temperatureRawBufPtr, *bloodPressRawBufPtr, *pulseRateRawBufPtr;
  unsigned short *measurementSelectionPtr;
  unsigned char *tempIndexPtr, *bloodPressIndexPtr, *pulseRateIndexPtr;
  bool *addFlagPtr;
}DataStructMeasure;
DataStructMeasure MeasureData;

//Task Compute's data
typedef struct DataStructCompute{
  unsigned char id;
  unsigned int *temperatureRawBufPtr, *bloodPressRawBufPtr, *pulseRateRawBufPtr;
  unsigned char *tempCorrectedBufPtr, *bloodPressCorrectedBufPtr;
  unsigned char *pulseRateCorrectedBufPtr;
  unsigned short *measurementSelectionPtr;
  unsigned char *tempIndexPtr, *bloodPressIndexPtr, *pulseRateIndexPtr;
  bool *addFlagPtr;
}DataStructCompute;
DataStructCompute ComputeData;

//Task Display's data
typedef struct DataStructDisplay{
  unsigned char id;
  unsigned char *tempCorrectedBufPtr, *bloodPressCorrectedBufPtr;
  unsigned char *pulseRateCorrectedBufPtr;
  unsigned short *batteryStatePtr;
  unsigned short *functionSelectPtr, *measurementSelectionPtr, *alarmAcknowledgePtr;
  unsigned char *tempIndexPtr, *bloodPressIndexPtr, *pulseRateIndexPtr;
  unsigned short *initial_val_menuPtr, *initial_val_AnnPtr;
  bool *addFlagPtr;
}DataStructDisplay;
DataStructDisplay DisplayData;

//Task WarningAlarm's data
typedef struct DataStructWarningAlarm{
  unsigned char id;
  unsigned int *temperatureRawBufPtr, *bloodPressRawBufPtr, *pulseRateRawBufPtr;
  unsigned short *batteryStatePtr, *alarmAcknowledgePtr;
  unsigned char *tempIndexPtr, *bloodPressIndexPtr, *pulseRateIndexPtr;
  bool *addFlagPtr, *addComFlagPtr;
}DataStructWarningAlarm;
DataStructWarningAlarm WarningAlarmData;

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

//Task TFTKeypad's data
typedef struct DataStructTFTKeypad{
  unsigned char id;
  unsigned short *functionSelectPtr, *measurementSelectionPtr, *alarmAcknowledgePtr;
  bool *addFlagPtr;
  unsigned short *initial_val_menuPtr, *initial_val_AnnPtr;
}DataStructTFTKeypad;
DataStructTFTKeypad TFTKeypadData;

//Task Communications's data
typedef struct DataStructCommunications{
  unsigned char id;
  unsigned char *tempCorrectedBufPtr, *bloodPressCorrectedBufPtr;
  unsigned char *pulseRateCorrectedBufPtr;
  unsigned short *batteryStatePtr;
  unsigned char *tempIndexPtr, *bloodPressIndexPtr, *pulseRateIndexPtr;
  bool *addFlagPtr;
}DataStructCommunications;
DataStructCommunications CommunicationsData;
```

Fig1 Control Flow diagram

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 orange and red. Last, the dynamic scheduler should add measurement in task queue every 5 second, and add communications task in the task queue when any warnings or alarms occur..

TEST SPECIFICATION:

- temperatureRawBuf, systolicPressRawBuf, diastolicPressRawBuf: Same simulation as last time but data is put in buffers.
- pulseRateRawBuf: Read in 0V to 3.3V square wave from function generator and use the external interrupt to count the number of pulses. Calculate the rate and put in the buffer.
- batteryStatus: Same as last time.
- A user interface with two pages on TFT
  - Menu: Selection for measurement
  - Annunciation: Show measurement values and acknowledgement button
- Blood pressure is shown orange when greater than 120 or less than 80, red when greater than upper bound 20%.
- Temperature is shown orange when less than 36.1 or greater than 37.8.

- Pulse rate is shown orange when less than 60 or greater than 100.
- A warning message is sent to the serial monitor when any warnings or alarms occur.
- Measurement is executed every 5 seconds by a dynamic scheduler.
- Each task is represented as a task control block written in a data struct with previous and next pointer.
- Data associated with each block is written as a data struct pointing to the required variables
- Dynamic scheduler uses double linked list to schedule the task queue.
- Major computation done on ATMega board with Uno board doing measurement.
- ATMega sends request for individual measurement to Uno and Uno sends back data to ATMega through serial communication.

TEST CASES:

- Show the task queue in serial monitor to verify the implementation of scheduler
- Show the corrected data on the TFT display
- Test the acknowledgement button on TFT

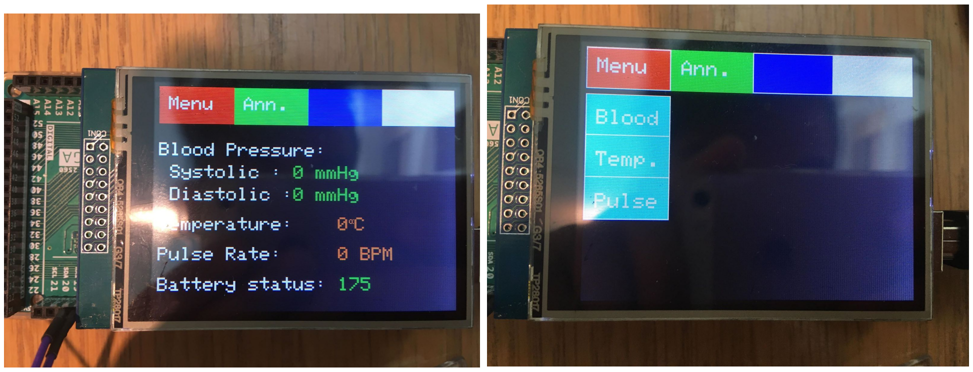
RESULT ANALYSIS:

Our runtime of each function in queue are listed in following table. We set the execution time for TFT keypad up to 800 millisecond to make the touch screen more roboust.

Table 1 Execution time

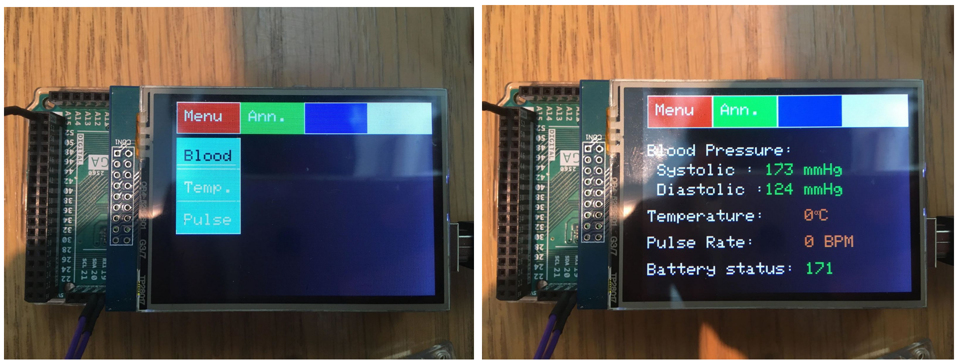
Communications	2 ms
Status	5 ms
Measure	6 ms
Compute	0 ms
WarningAlarm	0 ms
TFTKeypad	800 ms
Display	915 ms

The following picture shows our result of medical device user interface.



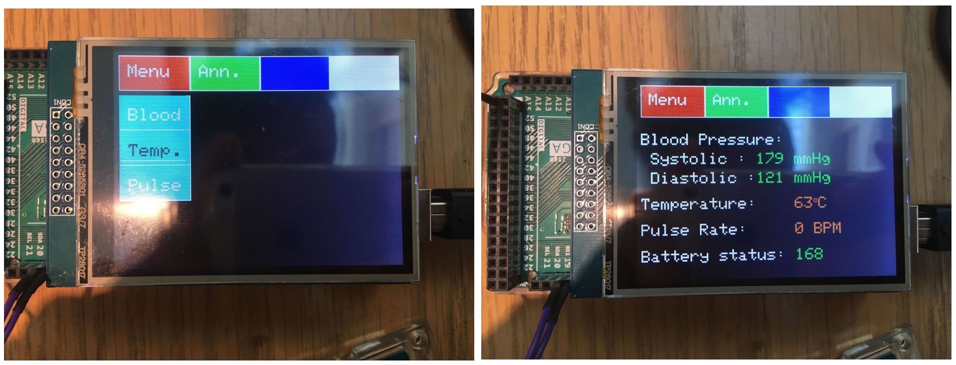
```
Blood Pressure:
 Systolic : 0 mmHg
 Diastolic :0 mmHg
Temperature:    0°C
Pulse Rate:     0 BPM
Battery status: 175
```

Fig 2 When no function pushed



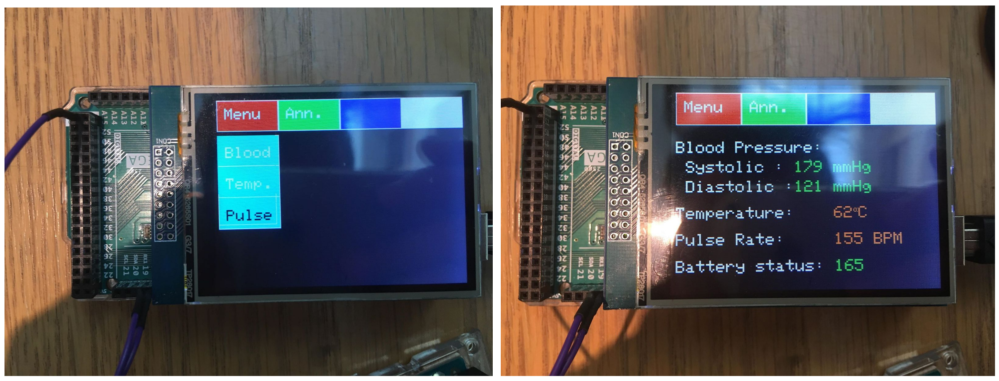
```
Blood Pressure:
 Systolic : 173 mmHg
 Diastolic :124 mmHg
Temperature:    0°C
Pulse Rate:     0 BPM
Battery status: 171
```

Fig 3 When blood function is pushed



```
Blood Pressure:
 Systolic : 179 mmHg
 Diastolic :121 mmHg
Temperature:    63°C
Pulse Rate:     0 BPM
Battery status: 168
```

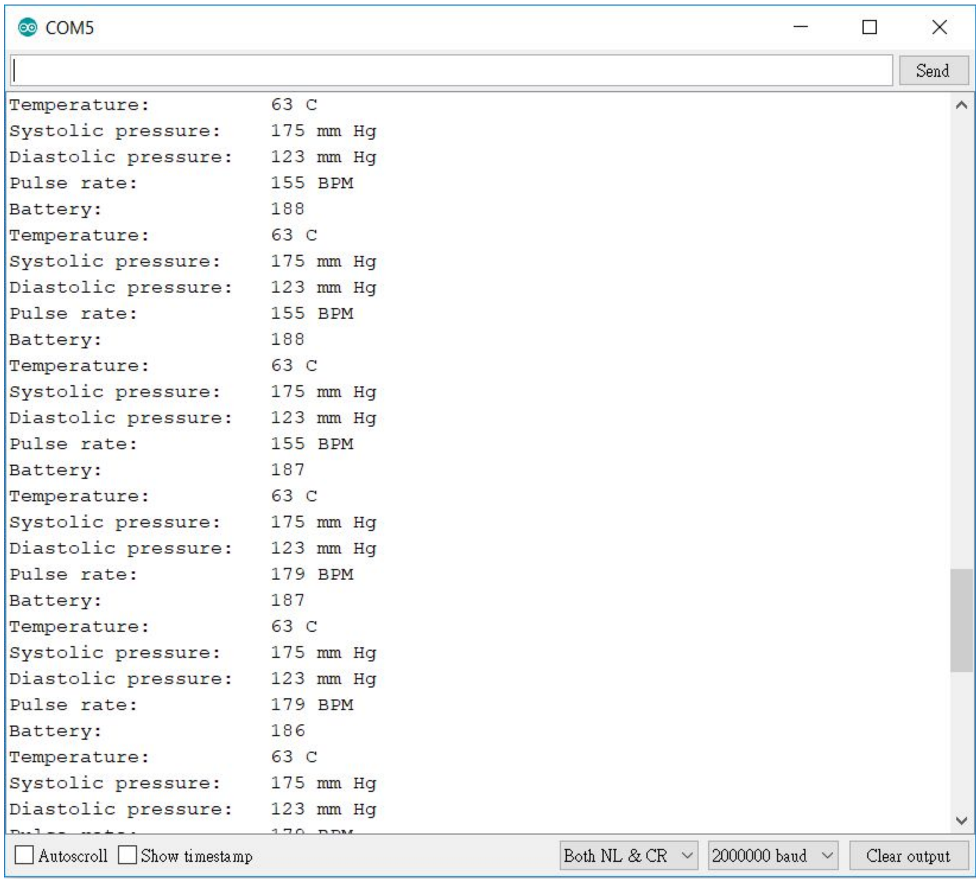
Fig 4 When Temp function is pushed



```
Blood Pressure:
 Systolic : 179 mmHg
 Diastolic :121 mmHg
Temperature:    62°C
Pulse Rate:     155 BPM
Battery status: 165
```

Fig 5 When Pulse function is pushed

When alarm occurs, the communication function will be activated and sent the corrected value to serial monitor.



COM5

```
Temperature:           63 C
Systolic pressure:     175 mm Hg
Diastolic pressure:    123 mm Hg
Pulse rate:            155 BPM
Battery:               188
Temperature:           63 C
Systolic pressure:     175 mm Hg
Diastolic pressure:    123 mm Hg
Pulse rate:            155 BPM
Battery:               188
Temperature:           63 C
Systolic pressure:     175 mm Hg
Diastolic pressure:    123 mm Hg
Pulse rate:            155 BPM
Battery:               187
Temperature:           63 C
Systolic pressure:     175 mm Hg
Diastolic pressure:    123 mm Hg
Pulse rate:            179 BPM
Battery:               187
Temperature:           63 C
Systolic pressure:     175 mm Hg
Diastolic pressure:    123 mm Hg
Pulse rate:            179 BPM
Battery:               186
Temperature:           63 C
Systolic pressure:     175 mm Hg
Diastolic pressure:    123 mm Hg
```

Fig 6 Warning message in serial monitor

ERROR ANALYSIS:

There are 4 error analysis case discussed below:

Memory Free problem

When implementing the insert and delete function used by scheduler, it is very important not to free the memory of deleted node but set the value to NULL. This problem cause us certain amount of time to debug.

Warning Acknowledge Button:

In this project, there is no way the text color for blood pressure turns to red due to the fact that data structure of unsigned char used in display could not store the value more than 255 while out of range value is 130 then co2*(130*1.2)+9 equal to 321. So we could not test our result on this part.

Pulse rate:

In this project, we misunderstood the description in the document so we used the same simulation method as the previous project. From the spec, it say we are going to use software internal interrupt. However, arduino does not have the ability to do software internal interrupt and that's why we think that this may mean we should still use software to simulate the pulse rate. Nevertheless, if we are going to use a hardware external interrupt, we can add a function generator to the arduino. The square wave input will add 1 to counter in the external interrupt function. When the pulse rate measurement task is called, the function will use the counter divided by the time interval as the current pulse rate.
Serial communication time:
In the previous project, we found that the serial communication time took about 1 sec to communicate. This is weird because we set a high baud rate, and the data is very trivial, the communication time shouldn't have take so long. Therefore, after surveying on the internet, we found that there is a function for serial port called "setTimeout()". The default value is set to 1 sec. After we change it to 50 msec, the communication took about 50 msec as we defined.

SUMMARY:

In this project, we practice building the task queue using double linked list. In each task, there will be some flags that is related to other task, which means that those task are only functioning when the flag is set to true, and will be delete from the queue after completed. To make the double link list work, we designed the schedule with insert and deleted function and counters that count the time for measurement last execution. For the measurement, we design a memory function that can store up to 8 historical data. We also practice to use TFT's keypad as user input.

CONCLUSION:

This project is a good practice toward a more reliable and efficient medical monitor system because it is equipped with dynamic task queue and external interrupt function. With the dynamic task queue, we can get the specific measurement we want directly without spending time on other measurement. On the other hand, with an external interrupt function, we can make the system adapt to some emergent situation. Also, the display and keypad tasks designed this time is a must for a monitor system product. By adding an acknowledge function, we allow the interaction between the system and the user. With all these task function, the project is very close to a final product.

CONTRIBUTION:

Alex:

- TCB define and initialize.
- Data struct for each TCB define and initialize.

- Design the dynamic scheduler
- List design spec and testing procedures.

Shawn:

- Design the TFT keypad touch mechanism and displays user interface.
- Construct the insert and delete queue function
- Elaborate error result analysis and error analysis in the project.

Shih-Hao:

- Designed the measurement task, compute task, and communication task
- Assisted the design of warning task
- Elaborate abstract, introduction, summary and conclusion in report
- Add error analysis for pulse rate and serial communication time

APPENDIX:

Mega

```
#include <Elegoo_GFX.h>    // Core graphics library
#include <Elegoo_TFTLCD.h> // Hardware-specific library
#include <TouchScreen.h>
#if defined(__SAM3X8E__)
   #undef __FlashStringHelper::F(string_literal)
   #define F(string_literal) string_literal
#endif

// 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).
//   D0 connects to digital pin 22
//   D1 connects to digital pin 23
//   D2 connects to digital pin 24
//   D3 connects to digital pin 25
//   D4 connects to digital pin 26
//   D5 connects to digital pin 27
//   D6 connects to digital pin 28
//   D7 connects to digital pin 29

// For the Arduino Due, use digital pins 33 through 40
// (on the 2-row header at the end of the board).
//   D0 connects to digital pin 33
//   D1 connects to digital pin 34
//   D2 connects to digital pin 35
//   D3 connects to digital pin 36
//   D4 connects to digital pin 37
//   D5 connects to digital pin 38
//   D6 connects to digital pin 39
//   D7 connects to digital pin 40
/*
#define YP 9  // must be an analog pin, use "An" notation!
#define XM 8  // must be an analog pin, use "An" notation!
#define YM A2   // can be a digital pin
#define XP A3   // can be a digital pin
*/

#define YP A3  // must be an analog pin, use "An" notation!
#define XM A2  // must be an analog pin, use "An" notation!
#define YM 9   // can be a digital pin
#define XP 8   // can be a digital pin
/*
#define TS_MINX 50
#define TS_MAXX 920

#define TS_MINY 100
#define TS_MAXY 940
*/
//Touch For New ILI9341 TP
#define TS_MINX 120
#define TS_MAXX 900

#define TS_MINY 70
```

```
#define TS_MAXY 920

// For better pressure precision, we need to know the resistance
// between X+ and X- Use any multimeter to read it
// For the one we're using, its 300 ohms across the X plate
TouchScreen ts = TouchScreen(XP, YP, XM, YM, 300);

#define LCD_CS A3
#define LCD_CD A2
#define LCD_WR A1
#define LCD_RD A0
// optional
#define LCD_RESET A4

// 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
#define ORANGE 0xFC00
#define W 80
#define H 40
#define PENRADIUS 3
#define Measure_Select_height 40
#define Measure_Select_width 80
#define MINPRESSURE 10
#define MAXPRESSURE 1000

Elegoo_TFTLCD tft(LCD_CS, LCD_CD, LCD_WR, LCD_RD, LCD_RESET);

/*Shared varibles*/
//Measurements
unsigned int temperatureRawBuf[8], bloodPressRawBuf[16];
unsigned int pulseRateRawBuf[8];
//Buffer index
```

```
unsigned char tempIndex=0, bloodPressIndex=0, pulseRateIndex=0;
//Display
unsigned char tempCorrectedBuf[8], bloodPressCorrectedBuf[16];
unsigned char pulseRateCorrectedBuf[8];
//Status
unsigned short batteryState=200;
//Alarms
unsigned char bpOutOfRange=0, tempOutOfRange=0, pulseOutOfRange=0;
//Warning
bool bpHigh=false, tempHigh=false, pulseLow=false;
//TFT Keypad
unsigned short functionSelect=0, measurementSelection=0, alarmAcknowledge=0;
unsigned short initial_val_menu=0, initial_val_Ann=0;

/*Define tasks*/
#define numTask 7
typedef struct TCB{
  void (*myTask)(void*);
  void *taskDataPtr;
  TCB *next=NULL, *prev=NULL;
}TCB;
String taskName[numTask] = {"Display", "TFTKeypad", "WarningAlarm", "Compute",
"Measure", "Status", "Communications"};
TCB Measure, Compute;
TCB Display, WarningAlarm, Status;
TCB TFTKeypad, Communications;
TCB *head=NULL, *tail=NULL, *currentTask=NULL;
TCB *taskArray[numTask];
bool taskAddFlag[numTask]={false};
bool taskInQue[numTask]={false};


/*Scheduler data*/
String message; //message of task time
#define taskqueFinishPin 30 //pin to be toggled after one cycle of task que
unsigned long mStart_time[4]={0,0,0,0}; //the start time of each measurement
bool mAvailable[4]={true,true,true,true}; //availibility of each measurement
unsigned long start_time; //the start time of each task
unsigned long taskTime[numTask]; //store the execution time of each task
```

```
/*Define data structures*/
//Task Measure's data
typedef struct DataStructMeasure{
  unsigned char id;
  unsigned int *temperatureRawBufPtr, *bloodPressRawBufPtr, *pulseRateRawBufPtr;
  unsigned short *measurementSelectionPtr;
  unsigned char *tempIndexPtr, *bloodPressIndexPtr, *pulseRateIndexPtr;
  bool *addFlagPtr;
}DataStructMeasure;
DataStructMeasure MeasureData;

//Task Compute's data
typedef struct DataStructCompute{
  unsigned char id;
  unsigned int *temperatureRawBufPtr, *bloodPressRawBufPtr, *pulseRateRawBufPtr;
  unsigned char *tempCorrectedBufPtr, *bloodPressCorrectedBufPtr;
  unsigned char *pulseRateCorrectedBufPtr;
  unsigned short *measurementSelectionPtr;
  unsigned char *tempIndexPtr, *bloodPressIndexPtr, *pulseRateIndexPtr;
  bool *addFlagPtr;
}DataStructCompute;
DataStructCompute ComputeData;

//Task Display's data
typedef struct DataStructDisplay{
  unsigned char id;
  unsigned char *tempCorrectedBufPtr, *bloodPressCorrectedBufPtr;
  unsigned char *pulseRateCorrectedBufPtr;
  unsigned short *batteryStatePtr;
  unsigned short *functionSelectPtr, *measurementSelectionPtr, *alarmAcknowledgePtr;
  unsigned char *tempIndexPtr, *bloodPressIndexPtr, *pulseRateIndexPtr;
  unsigned short *initial_val_menuPtr, *initial_val_AnnPtr;
  bool *addFlagPtr;
}DataStructDisplay;
DataStructDisplay DisplayData;

//Task WarningAlarm's data
```

```
typedef struct DataStructWarningAlarm{
  unsigned char id;
  unsigned int *temperatureRawBufPtr, *bloodPressRawBufPtr, *pulseRateRawBufPtr;
  unsigned short *batteryStatePtr, *alarmAcknowledgePtr;
  unsigned char *tempIndexPtr, *bloodPressIndexPtr, *pulseRateIndexPtr;
  bool *addFlagPtr, *addComFlagPtr;
}DataStructWarningAlarm;
DataStructWarningAlarm WarningAlarmData;

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

//Task TFTKeypad's data
typedef struct DataStructTFTKeypad{
  unsigned char id;
  unsigned short *functionSelectPtr, *measurementSelectionPtr, *alarmAcknowledgePtr;
  bool *addFlagPtr;
  unsigned short *initial_val_menuPtr, *initial_val_AnnPtr;
}DataStructTFTKeypad;
DataStructTFTKeypad TFTKeypadData;

//Task Communications's data
typedef struct DataStructCommunications{
  unsigned char id;
  unsigned char *tempCorrectedBufPtr, *bloodPressCorrectedBufPtr;
  unsigned char *pulseRateCorrectedBufPtr;
  unsigned short *batteryStatePtr;
  unsigned char *tempIndexPtr, *bloodPressIndexPtr, *pulseRateIndexPtr;
  bool *addFlagPtr;
}DataStructCommunications;
DataStructCommunications CommunicationsData;

void Insert(TCB* node){
   if(NULL==head){
        head=node;
        tail=node;

    }else{
        head->prev=node;
        node->next=head;
        head=head->prev;

    }
    return;
}

void Delete(TCB* node){

    if (head->next ==NULL){
        return;
    }else if (head==tail){
        head=tail=NULL;
    }else if(head==node){
        head= head->next;



    }else if (tail==node){
        tail=tail->prev;
    }else{
        node->prev->next=node->next;
        node->next->prev=node->prev;

    }
    return;


}


/*Tasks's function*/
void Measure_function(void *uncast_data){
  DataStructMeasure* data;
```

```
data = (DataStructMeasure*)uncast_data;
String  serialResponse;
char *p;
char *str;
int i = 0;
String cast_str;
int value[2];
char buf[10];

switch(*(data->measurementSelectionPtr)){
  // Temperature
  case 2:
    Serial1.write("1");
    while ( !Serial1.available()){}
    serialResponse = Serial1.readStringUntil('\n');
    *(data->tempIndexPtr) = (*(data->tempIndexPtr) + 1 ) % 8;
    *(data->temperatureRawBufPtr + *(data->tempIndexPtr)) = serialResponse.toInt();
    break;

  // Pressure
  case 1:
    Serial1.write("2");
    while ( !Serial1.available()){}
    serialResponse = Serial1.readStringUntil('\n');
    serialResponse.toCharArray(buf, sizeof(buf));
    p = buf;
    i = 0;

    while ((str = strtok_r(p, " ", &p)) != NULL) // delimiter is the semicolon
    {
      cast_str=(String)str;
      value[i] = cast_str.toInt();
      i++;
    }

    *(data->bloodPressIndexPtr) = (*(data->bloodPressIndexPtr) + 1 ) % 8;
    *(data->bloodPressRawBufPtr + *(data->bloodPressIndexPtr) ) = value[0];
    *(data->bloodPressRawBufPtr + *(data->bloodPressIndexPtr) + 8 ) = value[1];
    break;
    // Pulse rate
    case 3:
      Serial1.write("3");
      while ( !Serial1.available()){}
      serialResponse = Serial1.readStringUntil('\n');
      if(fabs(*(data->pulseRateRawBufPtr + *(data->pulseRateIndexPtr)) -
serialResponse.toInt())/(*(data->pulseRateRawBufPtr + *(data->pulseRateIndexPtr))) >= 0.15){
        *(data->pulseRateIndexPtr) = (*(data->pulseRateIndexPtr) + 1 ) % 8;
        *(data->pulseRateRawBufPtr + *(data->pulseRateIndexPtr)) = serialResponse.toInt();
      }
      break;

    default:
      // nothing
      break;
  }

  *(data->addFlagPtr) = false;
}

void Compute_function(void *uncast_data){
  DataStructCompute* data;
  data = (DataStructCompute *)uncast_data;

  switch(*(data->measurementSelectionPtr)){
    // Temperature
    case 2:
      *(data->tempCorrectedBufPtr + *(data->tempIndexPtr)) =
5+0.75*(*(data->temperatureRawBufPtr + *(data->tempIndexPtr)));
      break;

    // blood pressure
    case 1:
      *(data->bloodPressCorrectedBufPtr + *(data->bloodPressIndexPtr)) =
9+2*(*(data->bloodPressRawBufPtr + *(data->bloodPressIndexPtr)));
      *(data->bloodPressCorrectedBufPtr + *(data->bloodPressIndexPtr) + 8) =
6+1.5*(*(data->bloodPressRawBufPtr + *(data->bloodPressIndexPtr) + 8));
      break;
    // pulse rate
    case 3:
      *(data->pulseRateCorrectedBufPtr + *(data->pulseRateIndexPtr)) =
8+3*(*(data->pulseRateRawBufPtr + *(data->pulseRateIndexPtr)));
      break;

    default:
      break;
    // nothing

  }
  *(data->addFlagPtr) = false;
}

void Communications_function(void *uncast_data){
  DataStructCommunications* data;
  data=(DataStructCommunications*)uncast_data;

  Serial.write("Temperature:          ");
  Serial.print(*(data->tempCorrectedBufPtr + *(data->tempIndexPtr)));
  Serial.write(" C\n");
  Serial.write("Systolic pressure:    ");
  Serial.print(*(data->bloodPressCorrectedBufPtr + *(data->bloodPressIndexPtr)));
  Serial.write(" mm Hg\n");
  Serial.write("Diastolic pressure:   ");
  Serial.print(*(data->bloodPressCorrectedBufPtr + *(data->bloodPressIndexPtr) + 8));
  Serial.write(" mm Hg\n");
  Serial.write("Pulse rate:           ");
  Serial.print(*(data->pulseRateCorrectedBufPtr + *(data->pulseRateIndexPtr)));
  Serial.write(" BPM\n");
  Serial.write("Battery:              ");
  Serial.print(*(data->batteryStatePtr));
  Serial.write("\n");

  *(data->addFlagPtr) = false;
}

/*Helper function*/
```

```
//The upper bar text
void bar_text(){
  tft.setRotation(1);
  tft.setCursor(10, 10);
  tft.setTextSize(2);
  tft.setTextColor(WHITE,RED);
  tft.println("Menu");
  tft.setCursor(90, 10);
  tft.setTextColor(WHITE,GREEN);
  tft.print("Ann.");
  tft.setRotation(2);
 }


//The measurement function text without selection
void Measure_text(){
  tft.setRotation(1);
  tft.setCursor(10, 60);
  tft.setTextSize(2);
  tft.setTextColor(WHITE,CYAN);
  tft.println("Blood");
  tft.setCursor(10, 100);
  tft.print("Temp.");
  tft.setCursor(10, 140);
  tft.print("Pulse");
  tft.setRotation(2);
 }

//The measurement function 1 text
void Measure_text1(){
  tft.setRotation(1);
  tft.setCursor(10, 60);
  tft.setTextSize(2);
  tft.setTextColor(BLACK,CYAN);
  tft.println("Blood");
  tft.setRotation(2);
 }
```

```
//The measurement function 2 text
void Measure_text2(){
  tft.setRotation(1);
  tft.setTextSize(2);
  tft.setTextColor(BLACK,CYAN);
  tft.setCursor(10, 100);
  tft.print("Temp.");
  tft.setRotation(2);
 }
//The measurement function 2 text
void Measure_text3(){
  tft.setRotation(1);
  tft.setTextSize(2);
  tft.setTextColor(BLACK,CYAN);
   tft.setCursor(10, 140);
  tft.print("Pulse");
  tft.setRotation(2);
 }

//The text displayed in main page
void text_for_display(DataStructDisplay* data){
  tft.setRotation(1);
  tft.setCursor(0, 60);
  tft.setTextSize(2);
  tft.setTextColor(WHITE);
  tft.println("Blood Pressure: ");
  tft.setTextSize(1);
  tft.println("        ");
  tft.setTextSize(2);
  tft.print(" Systolic : ");
  if (bpOutOfRange==1){
    tft.setTextColor(ORANGE,BLACK);
    tft.print(*(data->bloodPressCorrectedBufPtr + *(data->bloodPressIndexPtr)));
    tft.println(" mmHg   ");}
  else if(*(data->alarmAcknowledgePtr)>5){
            tft.setTextColor(RED,BLACK);
    tft.print(*(data->bloodPressCorrectedBufPtr + *(data->bloodPressIndexPtr)));
    tft.println(" mmHg   ");
  }else{
    tft.setTextColor(GREEN,BLACK);
    tft.print(*(data->bloodPressCorrectedBufPtr + *(data->bloodPressIndexPtr)));
    tft.println(" mmHg   ");}};

  tft.setTextColor(WHITE);
  tft.setTextSize(1);
  tft.println("       ");
  tft.setTextSize(2);
  tft.print(" Diastolic :");
  if (bpOutOfRange==1){
    tft.setTextColor(ORANGE,BLACK);
    tft.print(*(data->bloodPressCorrectedBufPtr +8+ *(data->bloodPressIndexPtr)));
    tft.println(" mmHg ");}
  else{
    tft.setTextColor(GREEN,BLACK);
    tft.print(*(data->bloodPressCorrectedBufPtr + 8+*(data->bloodPressIndexPtr)));
    tft.println(" mmHg  ");}};

  tft.setTextColor(WHITE);
  tft.println("       ");
  tft.print("Temperature:    ");
  if (tempOutOfRange==1){tft.setTextColor(ORANGE,BLACK);}
  else{tft.setTextColor(GREEN,BLACK);};
  tft.print(*(data->tempCorrectedBufPtr + *(data->tempIndexPtr)));
  tft.setTextSize(1);
  tft.print((char)223);
  tft.setTextSize(2);
  tft.println("C  ");


  tft.setTextColor(WHITE);
  tft.println("       ");
  tft.print("Pulse Rate:     ");
  if (pulseOutOfRange==1){tft.setTextColor(ORANGE,BLACK);}
  else{tft.setTextColor(GREEN,BLACK);};
  tft.print(*(data->pulseRateCorrectedBufPtr + *(data->pulseRateIndexPtr)));
  tft.println(" BPM ");
```

```
   tft.setTextColor(WHITE);
   tft.println("        ");
   tft.print("Battery status: ");
   if (batteryState<=40){tft.setTextColor(ORANGE,BLACK);}
   else{tft.setTextColor(GREEN,BLACK);};
   tft.print(*(data->batteryStatePtr));
   tft.println("   ");

   tft.println();
   tft.println();
   tft.setRotation(2);


}

void Display_function(void *uncast_data){
  DataStructDisplay* data;
  data=(DataStructDisplay*)uncast_data;

  bar_text();



  if (*(data->functionSelectPtr)==0 )  {
    //Ann Select diagram first time enter
    if (*(data->initial_val_AnnPtr)==0){
    tft.setRotation(2);
    //tft.fillRect(0,320-2*W, H, W, GREEN);
    tft.fillRect(H+5,0, tft.width(),tft.height(), BLACK);
    *(data->initial_val_menuPtr)=0;
    bar_text();
    text_for_display(data);
    *(data->initial_val_AnnPtr)=1;
    }else{
            //Ann Select diagram second time enter
    bar_text();
    text_for_display(data);

      }
  //Todo the acknoledge block
  if (*(data->alarmAcknowledgePtr)>5)
                    *(data->alarmAcknowledgePtr)=0;
  //Serial.println("display screen");

}else{


  tft.setRotation(2);
  tft.drawRect(0,320-3*W, H, W, WHITE);
  for(int i=0; i<10000;i++){};
  //tft.drawRect(0,320-W, H, W, GREEN);

  if(*(data->measurementSelectionPtr)==1){

    //Menu Select diagram



    tft.drawRect(H+5,320-W+5, 1*H-5, W-5, WHITE);
    tft.drawRect(H+5,320-W+5, 2*H-5, W-5, CYAN);
    tft.drawRect(H+5,320-W+5, 3*H-5, W-5, CYAN);
    Measure_text();
    Measure_text1();



  }else if(*(data->measurementSelectionPtr)==2){

    //Menu Select diagram
    tft.drawRect(H+5,320-W+5, 2*H-5, W-5,  WHITE);
    tft.drawRect(H+5,320-W+5, 1*H-5, W-5, CYAN);
    tft.drawRect(H+5,320-W+5, 3*H-5, W-5, CYAN);
    Measure_text();
    Measure_text2();

  }else if(*(data->measurementSelectionPtr)==3){

    //Menu Select diagram
    tft.drawRect(H+5,320-W+5, 3*H-5, W-5, WHITE);
```

```
 tft.drawRect(H+5,320-W+5, 1*H-5, W-5, CYAN);
 tft.drawRect(H+5,320-W+5, 2*H-5, W-5, CYAN);
 Measure_text();
 Measure_text3();


}else {




 if (*(data->initial_val_menuPtr)==0){
 //tft.fillRect(0,320-2*W, H, W, GREEN);
 tft.fillRect(H+5,0, tft.width(),tft.height()-W, BLACK);
 tft.fillRect(4*H+5,0, tft.width(),tft.height(), BLACK);
 tft.fillRect(H+5,320-W, 1*H, W, CYAN);
 tft.fillRect(H+5,320-W, 2*H, W, CYAN);
 tft.fillRect(H+5,320-W, 3*H, W, CYAN);
 tft.drawRect(H+5,320-W, 1*H, W, WHITE);
 tft.drawRect(H+5,320-W, 2*H, W,  WHITE);
 tft.drawRect(H+5,320-W, 3*H, W, WHITE);
 bar_text();
 Measure_text();
 *(data->initial_val_menuPtr)=1;

 }else{

  Measure_text();
  }


 //text_for_display();

 }
 initial_val_Ann=0;
 }

}

void TFTKeypad_function(void *uncast_data){

 DataStructTFTKeypad * data;
 data=(DataStructTFTKeypad*)uncast_data;
 unsigned long start_time=millis();

 unsigned long count=0;
 while (count<800){
 digitalWrite(13, HIGH);
  TSPoint p = ts.getPoint();
  digitalWrite(13, LOW);

 // if sharing pins, you'll need to fix the directions of the touchscreen pins
 //pinMode(XP, OUTPUT);
  pinMode(XM, OUTPUT);
  pinMode(YP, OUTPUT);
 //pinMode(YM, OUTPUT);

 // we have some minimum pressure we consider 'valid'
 // pressure of 0 means no pressing!



 if (p.z > MINPRESSURE && p.z < MAXPRESSURE) {

  p.x = map(p.x, TS_MINX, TS_MAXX, tft.width(), 0);
  p.y = (tft.height()-map(p.y, TS_MINY, TS_MAXY, tft.height(), 0));

  /*Serial.print("("); Serial.print(p.x);
  Serial.print(", "); Serial.print(p.y);
  Serial.println(")");*/
```

```
tft.setRotation(2);

  if (p.x < H) {

    if (p.y < W) {
      //currentcolor = WHITE;
      tft.drawRect(0, 0, H, W, WHITE);
      *(data->functionSelectPtr)=-1;
    } else if (p.y < W*2) {
      //currentcolor = BLUE;

      tft.drawRect(0, W, H, W, WHITE);
      *(data->functionSelectPtr)=-1;
    } else if (p.y < W*3) {
      //currentcolor = GREEN;
      tft.drawRect( 0,W*2, H, W, WHITE);
      //*(data->Function_SelectPtr)=0;
      *(data->functionSelectPtr)=0;

    } else if (p.y < W*4) {
      //currentcolor = RED;
      tft.drawRect( 0,W*3, H, W, WHITE);
      //*(data->Function_SelectPtr)=1;
      *(data->functionSelectPtr)=1;
    }

  }


  if(*(data->functionSelectPtr)==0){

   if (p.x>H && p.x < 2*H && p.y> 2*W && p.y<3*W) {
     *(data->alarmAcknowledgePtr)=1;

   }

  }else if (p.y > tft.height()- Measure_Select_width){
        if (p.x>H && p.x<Measure_Select_height+H){
          *(data->measurementSelectionPtr)=1;

        }else if (p.x>H+Measure_Select_height && p.x<2*Measure_Select_height+H){
          *(data->measurementSelectionPtr)=2;


        }else if(p.x>H+2*Measure_Select_height && p.x<3*Measure_Select_height+H){
          *(data->measurementSelectionPtr)=3;

        }else if(p.x<H){
           *(data->measurementSelectionPtr)=0;

          }


      }
    }


  unsigned long end_time=millis();
  count=end_time-start_time;
  //Serial.print("count");
  //Serial.println(count);
  }

}


void WarningAlarm_function(void *uncast_data){
  DataStructWarningAlarm * data;
  data = ( DataStructWarningAlarm *)uncast_data;
  if(*(data->temperatureRawBufPtr + *(data->tempIndexPtr))<36.1 ||
*(data->temperatureRawBufPtr + *(data->tempIndexPtr))>37.8){
    tempOutOfRange=1;
      tempHigh=true;
    }else{
      tempHigh=false;
      tempOutOfRange=0;
  }
  if (*(data->bloodPressRawBufPtr + *(data->bloodPressIndexPtr))<120 ||
*(data->bloodPressRawBufPtr + *(data->bloodPressIndexPtr))>130){
      bpOutOfRange=1;
      *(data->addComFlagPtr) = true;
      if(*(data->bloodPressRawBufPtr + *(data->bloodPressIndexPtr))>130*1.2){
        *(data->alarmAcknowledgePtr) ++;
        bpHigh=true;
      }else{
        // reset acknowledge
        *(data->alarmAcknowledgePtr) = 0;
        bpHigh=false;
      }
    }else{
      // reset acknowledge
      *(data->alarmAcknowledgePtr) = 0;
      bpHigh=false;
      bpOutOfRange=0;
  }

  if (*(data->bloodPressRawBufPtr + *(data->bloodPressIndexPtr) + 8)<70 ||
*(data->bloodPressRawBufPtr + *(data->bloodPressIndexPtr) + 8)>80){
      bpOutOfRange=1;
    }else{
      bpOutOfRange=0;
  }

  if (*(data->pulseRateRawBufPtr + *(data->pulseRateIndexPtr))<60 ||
*(data->pulseRateRawBufPtr + *(data->pulseRateIndexPtr))>100){
      pulseOutOfRange=1;
      pulseLow=true;
    }else{
      pulseLow=false;
      pulseOutOfRange=0;

  }
  *(data->addFlagPtr) = false;
}
```

```
void Status_function(void *uncast_data){
  String  serialResponse;
  DataStructStatus* data;
  int value;
  data=(DataStructStatus *)uncast_data;
  Serial1.write("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 = cast_str.toInt();
  }
  //for(int j=0; j<1; j++) {

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

  *(data->batteryStatePtr)=value;

  *(data->addFlagPtr) = false;
}
```

```
void setup() {

  //Initialized buffers
  temperatureRawBuf[0] = 75; pulseRateRawBuf[0] = 0;
  bloodPressRawBuf[0] = 80; bloodPressRawBuf[8] = 80;

  //Initialized task Measure
  Measure.myTask = Measure_function;
  MeasureData.temperatureRawBufPtr = temperatureRawBuf;
  MeasureData.bloodPressRawBufPtr = bloodPressRawBuf;
  MeasureData.pulseRateRawBufPtr = pulseRateRawBuf;
  MeasureData.measurementSelectionPtr = &measurementSelection;
  MeasureData.tempIndexPtr = &tempIndex;
  MeasureData.bloodPressIndexPtr = &bloodPressIndex;
  MeasureData.pulseRateIndexPtr = &pulseRateIndex;
  MeasureData.id = 4;
  MeasureData.addFlagPtr = &taskAddFlag[4];
  Measure.taskDataPtr = &MeasureData;
  taskArray[4] = &Measure;

  //Initialized task Compute
  Compute.myTask = Compute_function;
  ComputeData.temperatureRawBufPtr = temperatureRawBuf;
  ComputeData.bloodPressRawBufPtr = bloodPressRawBuf;
  ComputeData.pulseRateRawBufPtr = pulseRateRawBuf;
  ComputeData.tempCorrectedBufPtr = tempCorrectedBuf;
  ComputeData.bloodPressCorrectedBufPtr = bloodPressCorrectedBuf;
  ComputeData.pulseRateCorrectedBufPtr = pulseRateCorrectedBuf;
  ComputeData.measurementSelectionPtr = &measurementSelection;
  ComputeData.tempIndexPtr = &tempIndex;
  ComputeData.bloodPressIndexPtr = &bloodPressIndex;
  ComputeData.pulseRateIndexPtr = &pulseRateIndex;
  ComputeData.id = 3;
  ComputeData.addFlagPtr = &taskAddFlag[3];
  Compute.taskDataPtr = &ComputeData;
  taskArray[3] = &Compute;

  //Initialized task Display
  Display.myTask = Display_function;
```

```
DisplayData.tempCorrectedBufPtr = tempCorrectedBuf;
DisplayData.bloodPressCorrectedBufPtr = bloodPressCorrectedBuf;
DisplayData.pulseRateCorrectedBufPtr = pulseRateCorrectedBuf;
DisplayData.batteryStatePtr = &batteryState;
DisplayData.measurementSelectionPtr = &measurementSelection;
DisplayData.alarmAcknowledgePtr = &alarmAcknowledge;
DisplayData.functionSelectPtr = &functionSelect;
DisplayData.tempIndexPtr = &tempIndex;
DisplayData.bloodPressIndexPtr = &bloodPressIndex;
DisplayData.pulseRateIndexPtr = &pulseRateIndex;
DisplayData.initial_val_menuPtr = &initial_val_menu;
DisplayData.initial_val_AnnPtr = &initial_val_Ann;
DisplayData.id = 0;
DisplayData.addFlagPtr = &taskAddFlag[0];
Display.taskDataPtr = &DisplayData;
taskArray[0] = &Display;

//Initialized task WarningAlarm
WarningAlarm.myTask = WarningAlarm_function;
WarningAlarmData.temperatureRawBufPtr = temperatureRawBuf;
WarningAlarmData.bloodPressRawBufPtr = bloodPressRawBuf;
WarningAlarmData.pulseRateRawBufPtr = pulseRateRawBuf;
WarningAlarmData.batteryStatePtr = &batteryState;
WarningAlarmData.alarmAcknowledgePtr = &alarmAcknowledge;
WarningAlarmData.tempIndexPtr = &tempIndex;
WarningAlarmData.bloodPressIndexPtr = &bloodPressIndex;
WarningAlarmData.pulseRateIndexPtr = &pulseRateIndex;
WarningAlarmData.id = 2;
WarningAlarmData.addFlagPtr = &taskAddFlag[2];
WarningAlarmData.addComFlagPtr = &taskAddFlag[6];
WarningAlarm.taskDataPtr = &WarningAlarmData;
taskArray[2] = &WarningAlarm;

//Initialized task Status
Status.myTask = Status_function;
StatusData.batteryStatePtr = &batteryState;
StatusData.id = 5;
StatusData.addFlagPtr = &taskAddFlag[5];
Status.taskDataPtr = &StatusData;
```

```
taskArray[5] = &Status;

//Initialized task TFTKeypad
TFTKeypad.myTask = TFTKeypad_function;
TFTKeypadData.measurementSelectionPtr = &measurementSelection;
TFTKeypadData.alarmAcknowledgePtr = &alarmAcknowledge;
TFTKeypadData.functionSelectPtr = &functionSelect;
TFTKeypadData.initial_val_menuPtr = &initial_val_menu;
TFTKeypadData.initial_val_AnnPtr = &initial_val_Ann;
TFTKeypadData.id = 1;
TFTKeypadData.addFlagPtr = &taskAddFlag[1];
TFTKeypad.taskDataPtr = &TFTKeypadData;
taskArray[1] = &TFTKeypad;

//Initialized task Communications
Communications.myTask = Communications_function;
CommunicationsData.tempCorrectedBufPtr = tempCorrectedBuf;
CommunicationsData.bloodPressCorrectedBufPtr = bloodPressCorrectedBuf;
CommunicationsData.pulseRateCorrectedBufPtr = pulseRateCorrectedBuf;
CommunicationsData.batteryStatePtr = &batteryState;
CommunicationsData.tempIndexPtr = &tempIndex;
CommunicationsData.bloodPressIndexPtr = &bloodPressIndex;
CommunicationsData.pulseRateIndexPtr = &pulseRateIndex;
CommunicationsData.id = 6;
CommunicationsData.addFlagPtr = &taskAddFlag[6];
Communications.taskDataPtr = &CommunicationsData;
taskArray[6] = &Communications;

//Initialized taskque
taskAddFlag[0] = true; Insert(taskArray[0]); taskInQue[0] = true;
taskAddFlag[1] = true; Insert(taskArray[1]); taskInQue[1] = true;
taskAddFlag[2] = true; Insert(taskArray[2]); taskInQue[2] = true;

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

//Initialized serial port 0 & 1
Serial.begin(2000000);
```

```
Serial1.begin(2000000);
Serial.setTimeout(5);
Serial1.setTimeout(5);

//Initialized for TFT
Serial.println(F("TFT LCD test"));
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 {
  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=0x9341;

}

tft.begin(identifier);
```

```
  tft.setRotation(2);

  tft.fillScreen(BLACK);

  tft.fillRect(0, 320-W, H, W, RED);
  //tft.fillRect(W, 0, W, W, YELLOW);
  tft.fillRect( 0,320-2*W, H, W, GREEN);
  //tft.fillRect(W*3, 0, W, W, CYAN);
  tft.fillRect(0,320-3*W, H, W, BLUE);
  //tft.fillRect(W*5, 0, W, W, MAGENTA);
  tft.fillRect(0,320-4*W, H, W, WHITE);

  tft.drawRect(0, 0, H, W, WHITE);

  pinMode(13, OUTPUT);
}

//taskArray = {"Display", "TFTKeypad", "WarningAlarm", "Compute", "Measure", "Status",
"Communications"}
/*Scheduler*/
void loop() {
  /*//set each task execution time to zero
  for (int i=0; i<numTask; i++)
    taskTime[i] = 0;*/
  // Enable Measure and Status tasks if time exceeds 5 seconds
  for (int i=0; i<4; i++){
    if (!mAvailable[i] && (millis() - mStart_time[i] >= 5000))
      mAvailable[i] = true;
  }
  // Set add flags of Status and WarningAlarm tasks when Status is available,
  // disable Status and start timer
  if (mAvailable[3]){
    taskAddFlag[5] = true;
    taskAddFlag[2] = true;
    mAvailable[3] = false;
    mStart_time[3] = millis();
  }
  // Set add flags of Measure, Compute and WarningAlarm tasks when selected Measure is
available,
  // disable the selected Measure and start timer
  if (measurementSelection)
    if (mAvailable[measurementSelection-1]){
      for (int i=0; i<3; i++)
        taskAddFlag[i+2] = true;
      mAvailable[measurementSelection-1] = false;
      mStart_time[measurementSelection-1] = millis();
  }
  // Sechdule the task by comparing the add flag and in queue flag of each task
  for (int i=0; i<numTask; i++){
    if (taskAddFlag[i]==false && taskInQue[i]==true){
      Delete(taskArray[i]);
      taskInQue[i] = false;
    }
    else if (taskAddFlag[i]==true && taskInQue[i]==false){
      Insert(taskArray[i]);
      taskInQue[i] = true;
    }
  }
  // Start executing the task queue
  currentTask = head;
  while (currentTask != NULL){
    //start_time = millis();
    (currentTask->myTask)(currentTask->taskDataPtr); //execute task
    //taskTime[*(unsigned char *)(currentTask->taskDataPtr)] = millis() - start_time;
    currentTask = currentTask->next;
  }
  //toggle pin after one cycle of task queue
  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());*/
}
```

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(2000000);
  Serial.setTimeout(5);
}

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

    // if receive 1, return measurement
    if( whichTask == "1" ){
       measureData = String(get_temperatureRaw());
       Serial.println(measureData);
    }
    else if( whichTask == "2" ){
      measureData = String(get_systolicPressRaw());
      measureData += " ";
      measureData += String(get_diastolicPressRaw());
      Serial.println(measureData);
    }
    else if( whichTask == "3" ){
      measureData = String(get_pulseRateRaw());
      Serial.println(measureData);
    }

    // if receive 4, return battery status
    else if( whichTask == "4" ){
      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;
  }

//  pulseRateRaw *= (1.0+(random(-20, 20)/100.0));
//  if( pulseRateRaw > 200) pulseRateRaw = 200;
//  else if( pulseRateRaw < 0) pulseRateRaw = 0;
//
//  // flip counter
//  pulseRateRaw_count *= -1 ;
  return pulseRateRaw;
}

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