Real-Time Environmental Monitoring and Control System

Real-Time Environmental Monitoring and Control System

The Real-Time Environmental Monitoring and Control System is a software project that aims to provide a comprehensive solution for monitoring and controlling various environmental parameters using a virtual device that simulates hardware sensors. The system consists of two main components: a control unit and a virtual device, which communicate with each other using RS-232 protocol and predefined command codes. The system offers a user-friendly graphical user interface (GUI) that enables users to interact with the system and view real-time and export the data from the sensors from log purpose . The system also supports continuous data streaming and data handling features.

Table of Contents

1. Communication Protocol

The communication protocol between the control unit and the virtual device is based on RS-232 serial communication standard and uses hex values to represent command and response codes. The messages are composed of one or more fields separated by a semicolon (;) character. Each message ends with a carriage return (\r) character.

The following table summarizes the command and response codes used by the system:

Code	Description
0x01	Request to List Sensors
0x02	Request Sensor Data
0x03	Start Data Stream with Interval
0x04	Stop Data Stream
0xA1	List of Sensors Response
0xA2	Sensor Data Response
0xA3	Data Stream Response

The following sections describe the format and examples of each message type.

Request to List Sensors

This message is sent by the control unit to request the list of sensors available on the virtual device. The message consists of only one field, which is the command code 0x01. The message ends with a carriage return (\r) character.

Example: 0x01\r

Request Sensor Data

This message is sent by the control unit to request the data from a specific sensor on the virtual device. The message consists of two fields: the command code 0x02 and the sensor ID. The sensor ID is a string that identifies the type and number of the sensor, such as T1 for temperature sensor 1 or H1 for humidity sensor 1. The message ends with a carriage return (\r) character.

Example: 0x02;T1\r

Start Data Stream with Interval

This message is sent by the control unit to start the data stream from the virtual device with a specified time interval between data transmissions. The message consists of two fields: the command code 0x03 and the interval parameter. The interval parameter is a 4-byte value that represents the time interval in milliseconds. The message ends with a carriage return (\r) character.

Example: 0x03;0x03E8\r

Stop Data Stream

This message is sent by the control unit to stop the data stream from the virtual device. The message consists of only one field, which is the command code 0x04. The message ends with a carriage return (\r) character>

Example: 0x04\r

List of Sensors Response

This message is sent by the virtual device in response to the request to list sensors from the control unit. The message consists of multiple fields: the response code 0xA1, followed by pairs of sensor type and sensor ID. The sensor type is a string that indicates the physical quantity measured by the sensor, such as Temperature or Humidity. The sensor ID is a string that identifies the type and number of the sensor, such as T1 for temperature sensor 1 or H1 for humidity sensor 1. The message ends with a carriage return (\r) character.

Example: 0xA1;Temperature;T1;Humidity;H1\r

Sensor Data Response

This message is sent by the virtual device in response to the request sensor data from the control unit. The message consists of three fields: the response code 0xA2, the sensor ID, and the sensor value. The sensor ID is a string that identifies the type and number of the sensor, such as T1 for temperature sensor 1 or H1 for humidity sensor 1. The sensor value is a floating-point number that represents the current reading of the sensor in the appropriate units, such as degrees Celsius or percentage. The message ends with a carriage return (\r) character.

Example: 0xA2;T1;25.3\r

Data Stream Response

This message is sent by the virtual device as part of the data stream to the control unit. The message consists of multiple fields: the response code 0xA3, followed by pairs of sensor ID and sensor value. The sensor ID is a string that identifies the type and number of the sensor, such as T1 for temperature sensor 1 or H1 for humidity sensor 1. The sensor value is a floating-point number that represents the current reading of the sensor in the appropriate units, such as degrees Celsius or percentage. The message ends with a carriage return (\r) character.

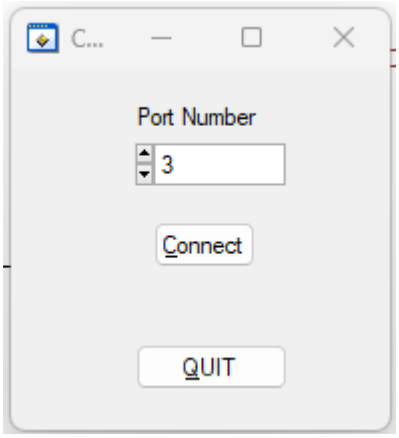
Example: 0xA3;T1;25.3;H1;60.2\r

2. Control Program

2.1. System Initialization

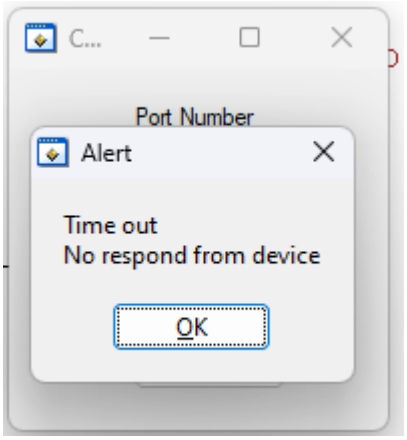
Open COM Port:

At the startup of the Control program the user will be ask to enter the port number that pair with the sensor device.
As explain in the communication section the control will send an command to request the list of the sensor.

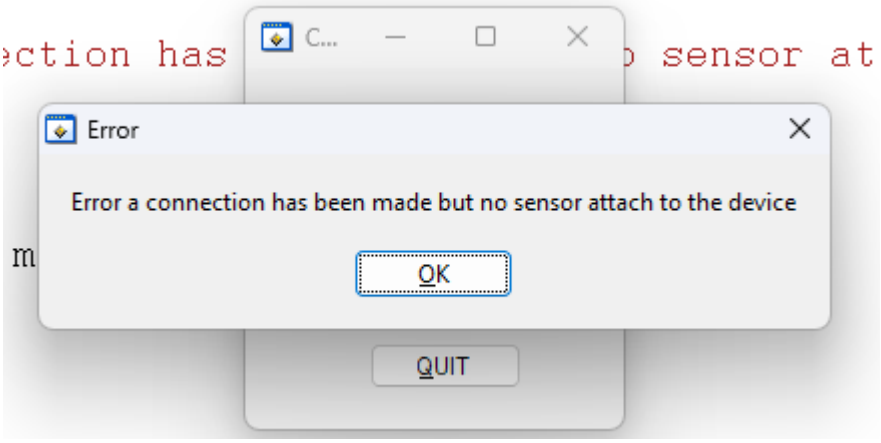


Handle Connection Errors:

Once the request for sensor list has been send the control will start a Watchdog on a separate thread.
Once the watchdog finish this count before a list has been send from the device.
the program will pop an error message:



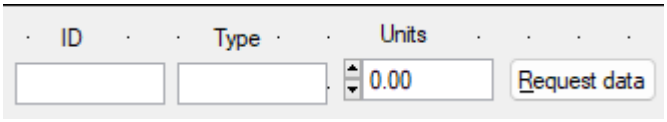
If the device send a respond but there are not sensor attach to it will prompt this massage:



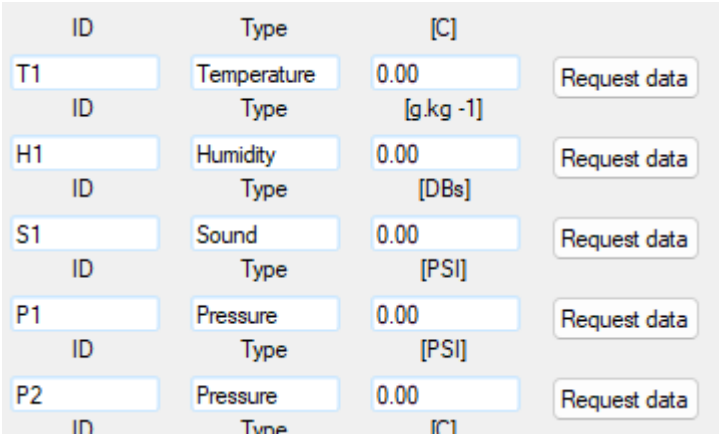
2.2. Build Main Program Body

Create Main UI:

Once the control receives the sensor list from the device it will build its UI according to list from this template:

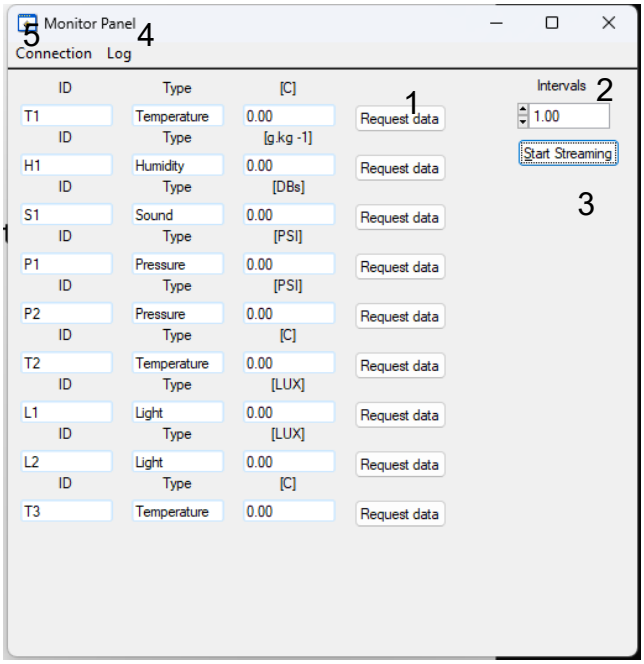


The ID show the ID(name) of the sensor.
The Type show the Type of the sensor.
Below the Unit show the latest data the it receive from the device.
The Unit will also display the correct unit according to sensor type like this:

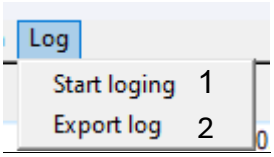


Each sensor has an Request data button that can request to update his data from the device.

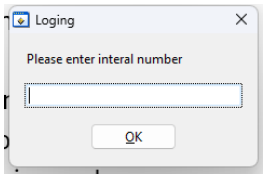
2.3. User guild

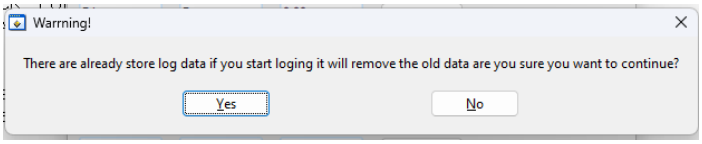


1. **<u>Request data:</u>** By pressing this button the control will send a request to the device with the sensor name.
2. **<u>Intervals:</u>** this indicates the time in seconds between each streaming that will be send by the device to the control.
3. **<u>Start streaming:</u>** By pressing this button the control will send a streaming request to the device at the intervals the show in (2). then it will turn it to “stop streaming” once press it will send a request to stop streaming to the device and turn back to it original function.
4. **<u>Log:</u>**



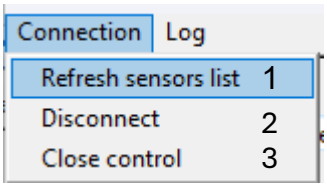
4.1. **<u>Start loging:</u>** By pressing this button the control will pop an massage request the user input to log the data from all the sensor every selected interval.
Then it will change to stop loging.
<u>*if there are already store log in the control this will overwrite the store log*</u>





4.2. **<u>Export log:</u>** will open a for the user to send the log data as an CSV file

5. **Connection:**



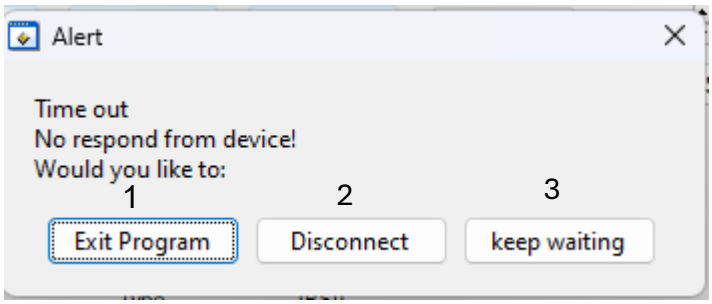
5.1. **Refresh sensors list: :** By pressing this button the control clear all the sensors from the control and send a request for sensor list, this action will recreate the control sensor.

5.2. **Disconnect:** By pressing this button, the control clear all the sensors from the control if it was streaming it will send a request to stop streaming and will return to the connection panel.

5.3. **Close control**: By pressing this button the control will exit the program. if it was streaming it will send a request to stop streaming.

2.4 Handline connection issue while the Monitor is on:

If the monitor sends a request for streaming and the watchdog counter finish this count before it receive a new data. The system will pop an error message:

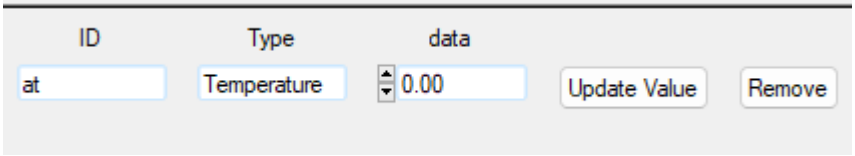


1. **Exit program:** this will close the control program.
2. **Disconnect**: this will disconnect and will reopen the connection panel.
3. **Keep waiting**: this will send another request for streaming and will restart the watchdog.

3. Emulator program:

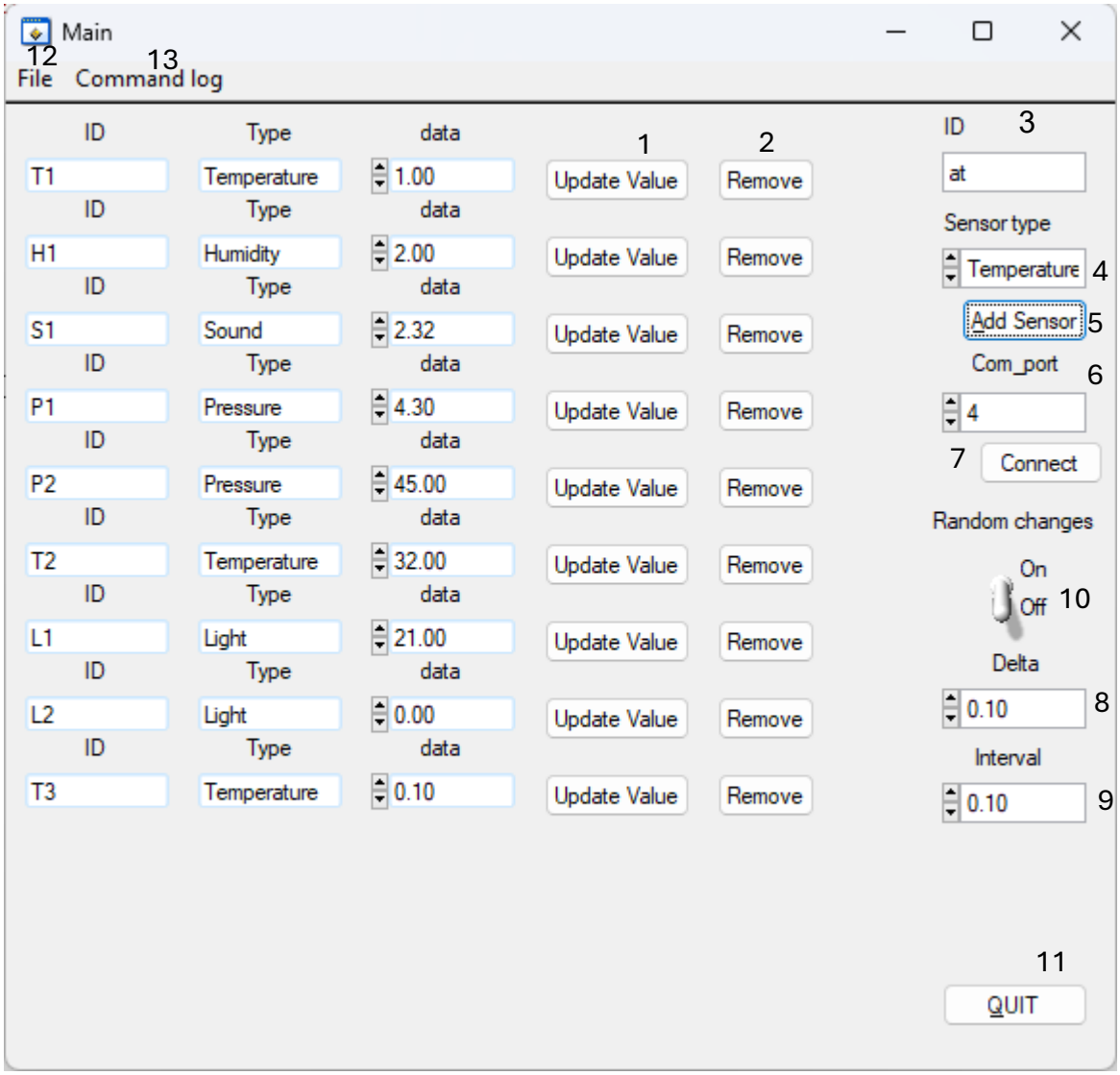
The emulator program is meant to act like the device that the sensor connected to it can add/remove sensor change their data.
it can export the existing template of the sensor and load a existing sensor from file.
it log the send and receive massage from and to the emulator. That can be export.

Then adding a sensor it will be add according to the template:



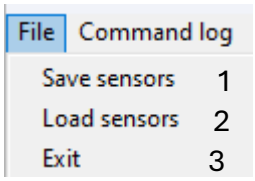
The ID show the ID(name) of the sensor.
The Type show the Type of the sensor.
Data the correct data of the sensor.
Update Value which will update the sensor data to the sensor.
Remove will remove the sensor.

3.1. Main panel - User Guild



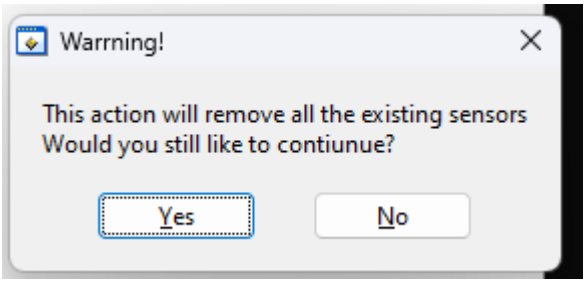
1. **Update Value:** This button will update the sensor data according to the correspond data filed
2. **Remove:** this button will remove the sensor from the list
3. **ID:** The ID name for the new sensor.
4. **Sensor Type:** the type for the new sensor.
5. **Add Sensor:** Add new sensor to the list of there is still space and if the ID wont conflict with another sensor, according to (3) and (4).
6. **Com_port:** the com port to connect to.
7. **Connect:** start the connect to the Com according to (6).
8. **Delta:** the Delta with a random value will be added to the sensor if (10) will be on : new_data(sensor)=+-delta*rand+data(sensor)
9. **Interval:** the interval which random change will be occurring
10. **Random changes:** toggle random change according to (8) and (9) information.
11. **QUIT:** close the program

12. **File:**

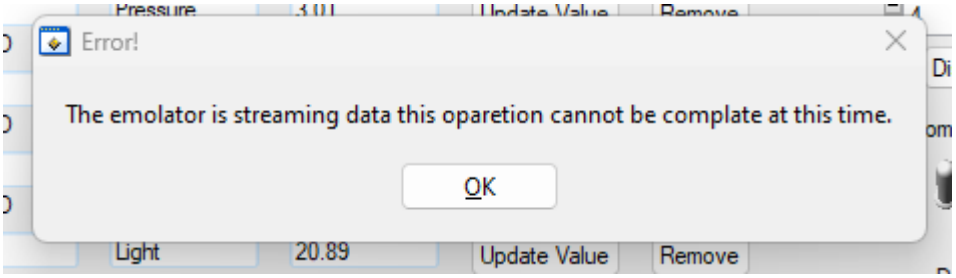


12.1. **Save sensors:** This will save the correct sensors and their correct data.

12.2. **Load sensors**: This will remove all the existing sensor and new load sensors from the template if there are existing sensors it will prompt this message:

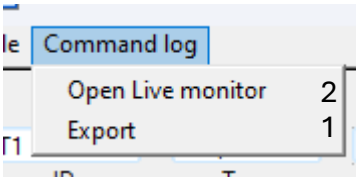


However, of the system is streaming it will not allow to load the sensor and will prompt this error:



12.3. **Exit:** this will close the emulator.

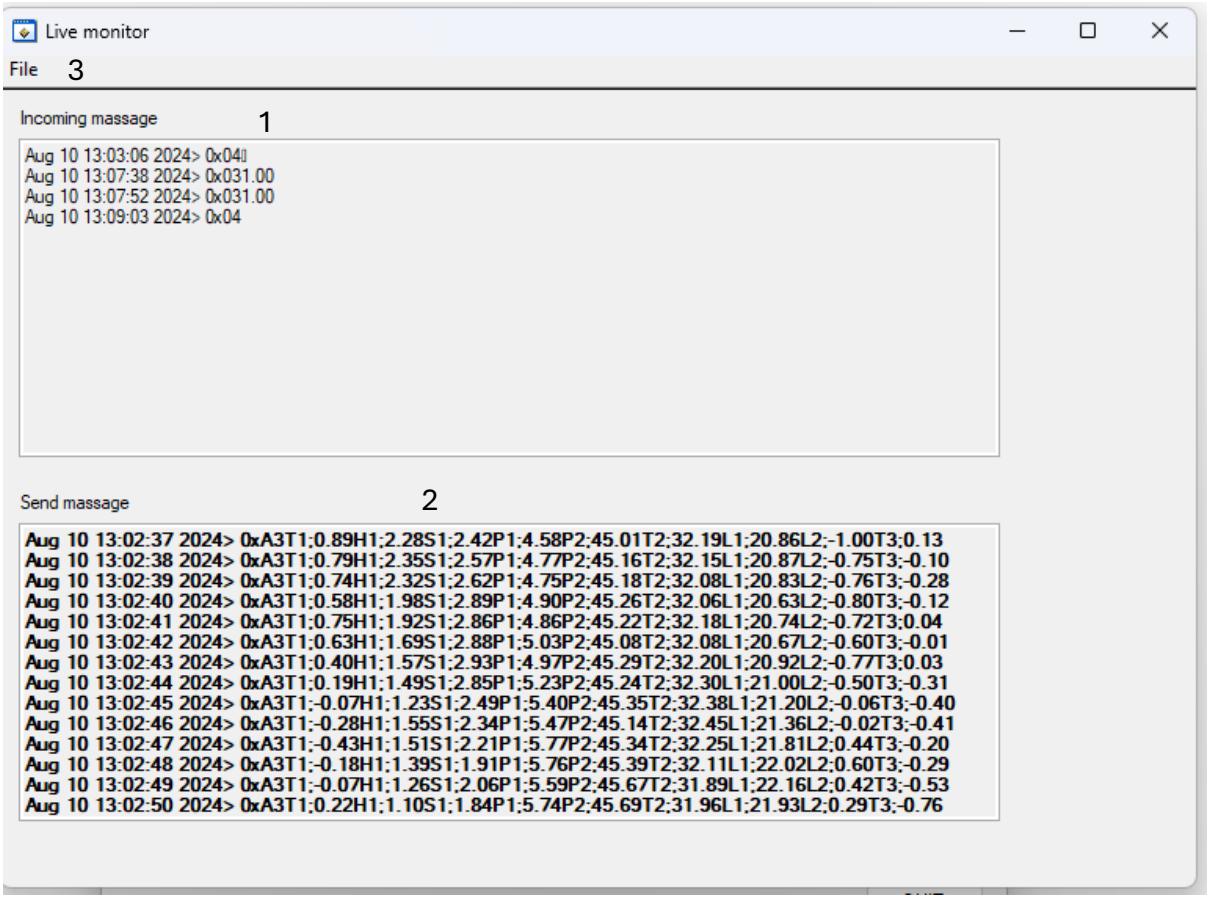
13. **Command log:**



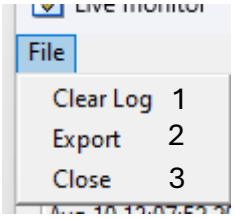
13.1. **Export:** open a save window to save the Send log and the receive log.

13.2. **Open live monitor:** open a send and receive log monitor window:

3.2. Live monitor panel – user guild



1. **Incoming massage:** this will show all the incoming massage from the control.
2. **Send massage:** this will show all the massage the sanded from the emulator.
3. **File:**



3.1. **Clear Log**: will clear all the send, receive log data.

3.2. **Export:** open a save window to save the Send log and the receive log.

3.3. **Close:** will close the log windows.

4. Algorithm

For the Algorithm I have made several DLL and a communication.h which held the communication protocol commands.

4.1. Sensor storage algorithm

For the sensor storage algorithm, I have created a struct for the sensor and a list node for the sensors, as well as several functions to use this struct, this have been done in sensor.dll which has been used in both control and emulator program.

The struct name for sensor is Sensor and the fields are:

- char id[MAX_SIZE_OF_ID] -> The name of each sensor
- SensorType type -> The type of each sensor
- double data -> the data for each sensor

The struct name for sensor list is Node and the fields are:

- Sensor *sensor;
- struct Node *next;

I have also define an enum with all the sensor type:

Name SensorType with the following :

TEMPERATURE_SENSOR, HUMIDITY_SENSOR, PRESSURE_SENSOR, GAS_SENSOR, LIGHT_SENSOR, SOUND_SENSOR

The Important function in the sensor DLL:

Sensor *initializeNewSensor(char *id, SensorType type);
Create a new Sensor with value of 0 and Type according to type.

Node *addSensor(Node *head, char *id, SensorType type);
Add new sensor to the list if the Head is NULL it will create the Head.
the function call to initializeNewSensor and it return the Head of the list.

void removeSensor(Node **head, char *id);
Remove the a sensor be the ID that it get from the list.

Sensor *findSensor(Node *head, char *id);
This function return the Sensor from the list if the ID if it cannot be found it return NULL.

4.2. Control unit algorithm

The control unit algorithm is the core of the system, as it manages the communication and data processing of the sensors. It consists of the following key components and functions:

The algorithm overview is as follows:

1. Initialization:
   The program starts by initializing the environment and loading the connection panel. It prompts the user to enter the port number that matches the sensor device. It creates locks for thread synchronization to ensure safe concurrent access to shared resources. It sends a command to the device to request the list of the sensors.
   create an struct for managing all the information in one place: base_info
2. Section Creation:
   To facilitate section creation, I developed a separate DLL (section_ctrl.dll) that includes functions for this purpose. Each section is handled by a structure (Section_ctrl) and managed through a node list called Section_Ctrl_Node.
3. Watchdog timer:
   A watchdog timer is implemented to check if there is a timeout so the system won't stuck while waiting for a respond from the device.
4. Logging:
   If logging is required the program maintains a log (dataLog) of activities or data, possibly sensor readings, and stores it in a large buffer.
5. Threading:
   The program uses threading (with WatchDogThread, LogingThread) to manage background tasks like logging and monitoring in case of a timeout.

4.2.1 Structs

Struct Section_ctrl fields:

- int inx -> the index of the section need in order to create them
- char id_sensor[MAX_SIZE_OF_ID] – the ID if the sensor which the section is attach to
- int id_ctrl -> the ctrl id of the sensor ID
- int type_ctrl -> the ctrl id of the sensor type
- int data_ctrl -> the ctrl id of the sensor data
- int requestUpdate_ctrl -> the ctrl id of the sensor button to request Update for the sensor

Struct Section_Ctrl_Node fields:

- Section_ctrl *section;
- struct Section_Ctrl_Node *next;

Struct base_info fields:

- int portnumber;
- Section_Ctrl_Node *sectionHead;
- Node *sensorHead;

4.2.2. Important Function from Connection_Panel.c and section_ctrl.c

From section_ctrl.c

int CVICALLBACK MyRequestUpdate
The callback function for request Update this will send a Request for data for a specific Sensor.

char *FindSectionIdByUpdateButton
Return the ID of the section there the UpdateButton were press.

int UpdateSectionData(int panel,base_info *mainInfo,char * id,Section_Ctrl_Node *cur_section);
This update the specific sensor that match the section data if cur_section is not NULL will save the process time to find the section with the match ID
This Function is taking the information from the sensor that match the ID and set them in the section.

int UpdateAllSectionData
This function update all the section from the sensor information

From Connection_Panel.c

static int WatchDog(void *task)
The WatcDog get an task that he need to watch for and start to count if he been stop before his count his complete his set the RetVal to 0, But if he completed the count he will set it to the massage task he got then he been called.

void AlertManager(Task task)
The alert Manager will preform the tast that it need
If an alert has been set if the stat is “waitForList” it will prompt that there is no device connected

If waitForMassage that means the system is waiting for a massage from the device but it didn't get one

int CVICALLBACK MyTimer
The timer callback is the backbone of the system.

- It keep track if an alarm has been set by the WatchDog, If there is an alart is will call to the AlertManager.
- Poll massage from the com port and decode them to the correct command according to communication.h and act according to the massage.

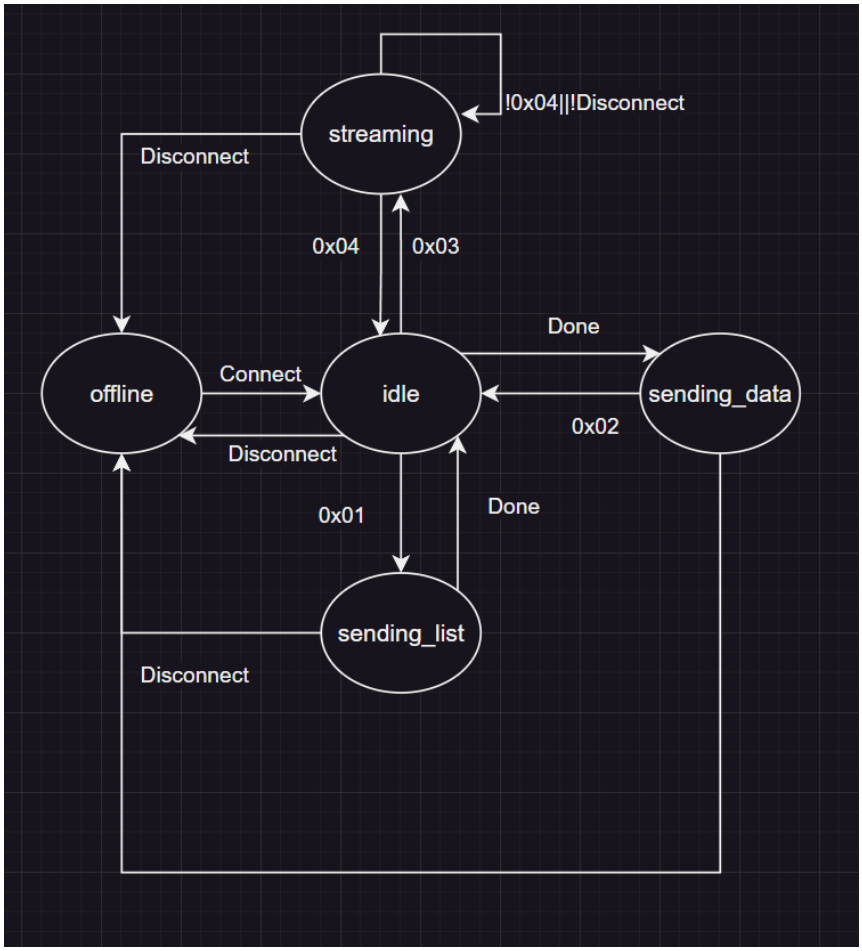
4.3. Emulator Algorithm

The emulator meant to act like a device then mean he got part the change the data and then it streaming it just sampling the sensor data at the given time

The algorithm overview is as follows:

1. Creating a sensor from a file or from the "Add sensor" button:
   To facilitate section creation, I developed a an struct that defines each section and several functions to manage them.
2. State machine structure:
   Since the emulator is meant to work as a device I have create it in a way the it has a state machine so it will know which operation it is working on now.
3. Loging the send and receive:
   Data then ever the emulator sends or receive data it will store in a buffer that can be view and save for monitoring.
4. Threading:
   The program uses threading for stream data to the control at the request intervals, this a sure that the streaming will continue even if the user is doing something with the emulator.

4.3.1 State machine:



4.3.2. Structs

Struct Section fields:

- int inx – the index for each Section
- char id_sensor – the ID for the sensor that related to the section
- int id_ctrl – the ctrl that held the ID in the section
- int type_ctrl - the ctrl that held the type as a string in the section
- int data_ctrl - the ctrl that held the data of the section sensor from there the update button will update to the sensor
- int updateButton_ctrl – the ctrl of the update button
- int removeButton_ctrl – the ctrl of the remove section button

SectionNode

- Section *section;
- struct SectionNode *next;

4.3.3. Important Function from emulator.c

int UpdateSensorFromSectionByInx(int panel, int inx,SectionNode *sectionhead,Node *sensorHead);
Update the sensor of the section with the match inx from the ctrl of the section.

void WriteLogToFile(const char* log, const char* filename)
Use to write a string to a file uses then save 2 Logs one for send and one for receive

void smControl(state next_state,char massage[])
The state machine function that managed the stats and calling the correct function to run the operation according to the correct state and next state it also receive an massage in case it needed to pass information.

static int streamingRun(void *intervales)
the thread function that streaming data according to the intervales that been send from the control unit.

int CVICALLBACK MyTimer
MyTimer is an important part of the emulator its decoding the massage and the command and responding with the correct state to call **smControl**

int CVICALLBACK MyRD
Start the random Timer function (int CVICALLBACK MyRDTimer) which changes the values of the sensors randomly.

void UpdateSectionData(SectionNode *sectionHead,int inx,double data)
Update the data of the section with the match inx with the set data.

5. Challenges and solutions.

1. **Organizing My Code:** When I first started with the creation of the emulator, I did not separate the sections into a distinct DLL, unlike in the control program. This led to significant confusion. To address this, I began the control program by first developing a DLL to handle the section structures. Furthermore, I implemented a struct to manage the main program details, covering the sensor head, section, and port numbers, ensuring these elements were directed to the DLL callback function for each dynamically generated section.
2.
3. **Communication Issues:** I faced numerous issues in facilitating communication between the programs. The callback installation failed; the program would freeze and couldn't read messages correctly. I resolved these problems by using timing polling instead of installing a callback function. To address the freezing issue, I employed “SetComTime ()” to prevent the function from freezing when attempting to read an empty buffer. I also tackled message reading problems through various methods. I created a header file containing connection information commands and protocols, using this same file in both programs. Additionally, I used “ComRdTerm()” with '\r' marking the message's end. For sending numbers, casting did not work initially, so after researching, I utilized the “memcpy()” function. At first, I was unable to read commands and compare them to the set commands in the header file, so I changed the type to unsigned char.
4. **Multithreading**
   Working with multithreading was difficult. First, I learned that the call thread can't modify the main panels. I needed a way to check if the watchdog had finished its job, so I used the “MyTimer()” callback function to monitor a shared variable for alerts. Sending data to all threads didn't work initially; I discovered that using dynamic memory allocation with malloc solved the issue. I also had to ensure I locked any variables accessed by multiple threads.

6. C source Code

6.1. sensors.c

```
#include <cvidef.h>
#include <ansi_c.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX_SIZE_OF_ID 10


DLLEXPORT int NUMBER_OF_SENSORS_TYPES = 6;

DLLEXPORT typedef enum {
   TEMPERATURE_SENSOR,
   HUMIDITY_SENSOR,
   PRESSURE_SENSOR,
   GAS_SENSOR,
   LIGHT_SENSOR,
   SOUND_SENSOR
} SensorType;

DLLEXPORT typedef struct {
   char id[MAX_SIZE_OF_ID];
   SensorType type;
   double data;
} Sensor;


DLLEXPORT typedef struct Node {
   Sensor *sensor;
```

```
  struct Node *next;
} Node;

Sensor* initializeNewSensor(char *id, SensorType type) {
        Sensor* newSensor=malloc(sizeof(Sensor));
        if (newSensor == NULL) {
    return NULL;
  }
  strncpy(newSensor->id, id, MAX_SIZE_OF_ID*sizeof(char));
  newSensor->type = type;
  newSensor->data = 0.0;
        return newSensor;
}
Node* addSensor(Node *head, char *id, SensorType type) {
        Node *newNode = malloc(sizeof(Node));

        if(newNode==NULL) return NULL; //Cannot allocate memory for new newNode
        newNode->sensor=initializeNewSensor(id, type);
        if(newNode->sensor==NULL) return NULL; //Cannot allocate memory for new
sensor
        newNode->next=NULL;
        if(head==NULL) head=newNode;
        else
        {
                Node *tmpNode=head;
                while(tmpNode->next!=NULL) tmpNode=tmpNode->next;
                tmpNode->next=newNode;

        }
  return head;
}
```

```
void freelist(Node *head)
{
        if(head!=NULL)
        {
                freelist(head->next);
                free(head->sensor);
                free(head);
        }
}

void removeSensor(Node **head, char *id) {
  Node *current = *head;
  Node *previous = NULL;

  while (current != NULL && strcmp(current->sensor->id, id) != 0) {
    previous = current;
    current = current->next;
  }

  if (current == NULL) {
    return ;
  }

  if (previous == NULL) {
    *head = current->next;
  } else {
    previous->next = current->next;
  }
        free(current->sensor);
  free(current);
```

```
    return;
}

Sensor* findSensor(Node *head, char *id) {
    Node *current = head;
    while (current != NULL) {
        if (strcmp(current->sensor->id, id) == 0) {
            return current->sensor;
        }
        current = current->next;
    }
    return NULL;
}

double getSensorData(Node *head, char *id) {
    Sensor *sensor = findSensor(head, id);
    if (sensor != NULL) {
        return sensor->data;
    }
    return DBL_MIN; // Indicate that the sensor was not found return minimum value of a
double
}

int setSensorData(Node *head, char *id,double data) {
    Sensor *sensor = findSensor(head, id);
    if (sensor != NULL) {
                    sensor->data=data;
        return 0;
    }
    return -1; // Indicate that the sensor was not found return -1
}
```

```
SensorType getSensorType(Node *head, char *id) {
   Sensor *sensor = findSensor(head, id);
   if (sensor != NULL) {
      return sensor->type;
   }
   return sensor->type; // Indicate that the sensor was not found return minimum value of a double
}

char* getSensorTypeString(SensorType type) {
   switch (type) {
      case TEMPERATURE_SENSOR:
         return "Temperature";
      case HUMIDITY_SENSOR:
         return "Humidity";
      case PRESSURE_SENSOR:
         return "Pressure";
      case GAS_SENSOR:
         return "Gas";
      case LIGHT_SENSOR:
         return "Light";
      case SOUND_SENSOR:
         return "Sound";
   }
          return "Unknown";
}

char* getSensorTypeUnit(SensorType type)
{
   switch (type) {
```

```
      case TEMPERATURE_SENSOR:
        return "[C]";
      case HUMIDITY_SENSOR:
        return "[g.kg -1]";
      case PRESSURE_SENSOR:
        return "[PSI]";
      case GAS_SENSOR:
        return "[Gas]";
      case LIGHT_SENSOR:
        return "[LUX]";
      case SOUND_SENSOR:
        return "[DBs]";
   }
          return "Unknown";
}
```

6.2. sensor.h

```
#include <cvidef.h>
#define MAX_SIZE_OF_ID 10

/************** Static Function Declarations **************/
/************** Global Variable Declarations **************/
extern DLLIMPORT int NUMBER_OF_SENSORS_TYPES;

DLLEXPORT typedef enum {
   TEMPERATURE_SENSOR,
   HUMIDITY_SENSOR,
   PRESSURE_SENSOR,
   GAS_SENSOR,
   LIGHT_SENSOR,
   SOUND_SENSOR
```

```
} SensorType;

DLLEXPORT typedef struct {
   char id[MAX_SIZE_OF_ID];
   SensorType type;
   double data;
} Sensor;


DLLEXPORT typedef struct Node {
   Sensor *sensor;
   struct Node *next;
} Node;

/************** Global Function Declarations **************/
extern Sensor *initializeNewSensor(char *id, SensorType type);
extern Node *addSensor(Node *head, char *id, SensorType type);
extern void freelist(Node *head);
extern void removeSensor(Node **head, char *id);
extern Sensor *findSensor(Node *head, char *id);
extern double getSensorData(Node *head, char *id);
extern int setSensorData(Node *head, char *id, double data);
extern char *getSensorTypeString(SensorType type);
extern SensorType getSensorType(Node *head, char *id);
extern char *getSensorTypeUnit(SensorType type);
```

6.3. section_ctrl.c

```
#include <utility.h>
#include <rs232.h>
#include <ansi_c.h>

#include <userint.h>

#include "communication.h"
#include "section_ctrl.h"



//////Function Define//////////////////

Section_ctrl* initializeNewSection(int panel,char *id,int inx,base_info *mainInfo);
int CreateSections(int panel,Section_Ctrl_Node **sectionHead,base_info *mainInfo);
int UpdateSectionData(int panel,base_info *mainInfo,char * id,Section_Ctrl_Node
*cur_section);
char *FindSectionIdByUpdateButton(base_info *mainInfo, int requestUpdate_ctrl);
int UpdateAllSectionData(int panel,base_info *mainInfo);
void freeCtrlSection(int panel,Section_Ctrl_Node *sectionHead);
int CVICALLBACK MyRequestUpdate (int panel, int control, int event,void
*callbackData, int eventData1, int eventData2);




////////////////////////////////////



///////Global parameters//////////////////

static int numOfSections=0;
```

```
int (__cdecl *ptrMyRequestUpdate)(int, int, int, void *, int, int) =  MyRequestUpdate;



//////////////////////////////////////



Section_ctrl* initializeNewSection(int panel,char *id,int inx,base_info *mainInfo) {
        if(inx>MAX_SECTIONS) return NULL;
        Node *sensorHead=mainInfo->sensorHead;
        Section_ctrl* newSection=malloc(sizeof(Section_ctrl));
        if (newSection == NULL) {
    return NULL;
  }
  strncpy(newSection->id_sensor, id, MAX_SIZE_OF_ID*sizeof(char));
  newSection->inx = inx;
  // Create controls dynamically


  newSection->id_ctrl = NewCtrl(panel, CTRL_STRING, "ID", inx *
DISTANT_BETWEEN_SECTION+DISTANT_FROM_TOP, DISTANT_FROM_LEFT);
  newSection->type_ctrl = NewCtrl(panel, CTRL_STRING, "Type", inx *
DISTANT_BETWEEN_SECTION+DISTANT_FROM_TOP,
DISTANT_FROM_LEFT+DISTANT_BETWEEN_CTRL);
        newSection->data_ctrl = NewCtrl(panel, CTRL_NUMERIC,
getSensorTypeUnit(getSensorType(sensorHead,id)), inx *
DISTANT_BETWEEN_SECTION+DISTANT_FROM_TOP,
DISTANT_FROM_LEFT+DISTANT_BETWEEN_CTRL*2);
  newSection->requestUpdate_ctrl = NewCtrl(panel,
CTRL_SQUARE_COMMAND_BUTTON, "Request data", inx *
DISTANT_BETWEEN_SECTION+DISTANT_FROM_TOP,
DISTANT_FROM_LEFT+DISTANT_BETWEEN_CTRL*3);


        //set control mode and other setting for each ctrl
        SetCtrlAttribute (panel,newSection->id_ctrl , ATTR_CTRL_MODE, 0);
        SetCtrlAttribute (panel,newSection->type_ctrl , ATTR_CTRL_MODE, 0);
```

```
	SetCtrlAttribute (panel,newSection->data_ctrl , ATTR_CTRL_MODE, 0);

	SensorType tmptype=getSensorType(sensorHead,id);

	char* type_str=getSensorTypeString(tmptype);

	SetCtrlVal (panel, newSection->type_ctrl, type_str);

	SetCtrlVal (panel, newSection->id_ctrl, id);

	SetCtrlAttribute (panel, newSection->requestUpdate_ctrl,
ATTR_CALLBACK_FUNCTION_POINTER, ptrMyRequestUpdate);

	SetCtrlAttribute (panel, newSection->requestUpdate_ctrl,
ATTR_CALLBACK_DATA, (void*)mainInfo);


	return newSection;

}


int CreateSections(int panel,Section_Ctrl_Node **sectionHead,base_info *mainInfo)

{

	Node *sensorHead=mainInfo->sensorHead;

	numOfSections=0;

	Section_Ctrl_Node *current = *sectionHead;

	Node *correctSensor=sensorHead;

	while(correctSensor!=NULL&&numOfSections<MAX_SECTIONS)

	{

		current->section=initializeNewSection(panel,correctSensor->sensor-
>id,numOfSections,mainInfo);

		if(correctSensor->next!=NULL)

		{

			current->next=(Section_Ctrl_Node*)
malloc(sizeof(Section_Ctrl_Node));

			if(current->next==NULL) return -2;// error code for fail to allocat
memory

			current=current->next;

		}else current->next=NULL;
```

```
			numOfSections++;
			correctSensor=correctSensor->next;
		}
		if(numOfSections==MAX_SECTIONS) return -1;
		return numOfSections;
}

void freeCtrlSection(int panel,Section_Ctrl_Node *sectionHead)
{
		if(sectionHead!=NULL)
		{
			freeCtrlSection(panel,sectionHead->next);
			DiscardCtrl (panel, sectionHead->section->id_ctrl);
			DiscardCtrl (panel, sectionHead->section->data_ctrl);
			DiscardCtrl (panel, sectionHead->section->requestUpdate_ctrl);
			DiscardCtrl (panel, sectionHead->section->type_ctrl);
			free(sectionHead->section);
			free(sectionHead);
		}else numOfSections=0;
}

int UpdateSectionData(int panel,base_info *mainInfo,char * id,Section_Ctrl_Node
*cur_section)//if cur_section =NULL look for the correct section
{
		Node *sensorHead=mainInfo->sensorHead;
		Section_Ctrl_Node *sectionHead= mainInfo->sectionHead;
		double data=getSensorData(sensorHead,id);
		if(data==-1) return -2; // cannot find sensor for this id
		if(cur_section!=NULL)
		{
			SetCtrlVal(panel,cur_section->section->data_ctrl,data);
```

```
		}
		else
		{
			Section_Ctrl_Node *current = sectionHead;
			while(current !=NULL&&(strcmp(current->section->id_sensor, id) != 0))
current=current->next;
			if(current==NULL) return -1;// cannot find section for this sensor
			SetCtrlVal(panel,current->section->data_ctrl,data);
		}
		return 0;
}

char *FindSectionIdByUpdateButton(base_info *mainInfo, int requestUpdate_ctrl) {
		Node *sensorHead=mainInfo->sensorHead;
		Section_Ctrl_Node *sectionHead= mainInfo->sectionHead;
		Section_Ctrl_Node *current =sectionHead;
  while (current != NULL) {
    if (current->section->requestUpdate_ctrl == requestUpdate_ctrl) {
      return current->section->id_sensor;
    }
    current = current->next;
  }
  return NULL; // Indicate that the section was not found
}

int UpdateAllSectionData(int panel,base_info *mainInfo)
{
		Node *sensorHead=mainInfo->sensorHead;
		Section_Ctrl_Node *sectionHead= mainInfo->sectionHead;
		Section_Ctrl_Node *current = sectionHead;
		int err=0;
```

```
	while(current!=NULL)
	{
		err=UpdateSectionData(panel,mainInfo,current->section->id_sensor,current);
		current=current->next;
	}
	return err;
}

int CVICALLBACK MyRequestUpdate (int panel, int control, int event,void *callbackData, int eventData1, int eventData2)
{
	switch (event)
	{
		case EVENT_COMMIT:
			base_info *mainInfo=(base_info*)callbackData;
			char *id=FindSectionIdByUpdateButton(mainInfo,control);
			if(id==NULL) return;
			char massage[SEND_BUFFER_SIZE];
			//massage[0]=Request_Sensor_Data;
			//massage[1]=0;
			sprintf(massage,"%c%s",Request_Sensor_Data,id);
			int a=strlen(massage);
			massage[a++]=0;
			massage[a++]='\r';
			ComWrt (mainInfo->portnumber, massage, a);
		break;
	}
	return 0;
}
```

6.4. section_ctrl.h

```
#include <cvirte.h>
#include "sensors.h"
#define MAX_SIZE_OF_ID 10
#define MAX_SECTIONS 11
#define DISTANT_BETWEEN_SECTION 40
#define DISTANT_BETWEEN_CTRL 90
#define DISTANT_FROM_LEFT 10
#define DISTANT_FROM_TOP 50



/************** Static Function Declarations **************/

/************** Global Variable Declarations **************/



/************** Global Function Declarations **************/



////////////////////////////////////////////////
////////////////Define Struct/////////////////////
////////////////////////////////////////////////

DLLEXPORT typedef struct {
        int inx;
        char id_sensor[MAX_SIZE_OF_ID];
  int id_ctrl;
        int type_ctrl;
```

```
   int data_ctrl;
   int requestUpdate_ctrl;
} Section_ctrl;



DLLEXPORT typedef struct Section_Ctrl_Node{
        Section_ctrl *section;
   struct Section_Ctrl_Node *next;
} Section_Ctrl_Node;

DLLEXPORT typedef struct {
        int portnumber;
        Section_Ctrl_Node *sectionHead;
        Node *sensorHead;
} base_info;



//////////////////////////////////////////////

extern Section_ctrl* initializeNewSection(int panel,char *id,int inx,base_info
*mainInfo);
extern int CVICALLBACK MyRequestUpdate (int panel, int control, int event,void
*callbackData, int eventData1, int eventData2);
extern int CreateSections(int panel,Section_Ctrl_Node **sectionHead,base_info
*mainInfo);
extern int UpdateSectionData(int panel,base_info *mainInfo,char *
id,Section_Ctrl_Node *cur_section);
extern char *FindSectionIdByUpdateButton(base_info *mainInfo, int
requestUpdate_ctrl);
extern int UpdateAllSectionData(int panel,base_info *mainInfo);
extern void freeCtrlSection(int panel,Section_Ctrl_Node *sectionHead);
```

6.5. communication.h

```
#include <cvidef.h>
#include <ansi_c.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

//////////////////////////
/////Define Commandes///////
#define Request_to_List_Sensors 0x01
#define Request_Sensor_Data 0x02
#define Start_Data_Stream 0x03
#define Stop_Data_Stream 0x04
#define List_of_Sensors_Response 0xA1
#define Sensor_Data_Response 0xA2
#define Data_Stream_Response 0xA3
////////////////////////////////
/////Define Comunication protocal//////
#define Baud_Rate 256000
#define Parity 0
#define Data_Bits 8
#define Stop_Bits 1
#define SEND_BUFFER_SIZE 32767
#define MAX_WAIT_TIME 1000 //maximun wait if the massage didnt get /r in mile sec
//////////////////////////
```

6.6. Connection_Panel.c

```
#include <rs232.h>
#include <cvirte.h>
#include <userint.h>
#include <cvirte.h>
#include <utility.h>
#include "Connection_Panel.h"
#include "Monitor_panel.h"
#include "communication.h"
#include "section_ctrl.h"

#define MAX_LOG_LENGHT 100000
#define BUFFER_SIZE 200

typedef enum{
        Nothing,
        waitForList,
        waitForMassage,
} Task;



void UpdateSensorsSections(char massage[],int l);
void UpdateSensorsData(char massage[],int l);
int SendReqForSensors();
int CVICALLBACK MyStopStreaming (int panel, int control, int event,void *callbackData,
int eventData1, int eventData2);
int CVICALLBACK MyStartStreaming (int panel, int control, int event,void *callbackData,
int eventData1, int eventData2);
void ResetWatchDog();
void StopWatchDog();
```

```
void StartWatchDog(Task task);
static int WatchDog(void *task);
void AlertManager(Task task);
void GetCurrentTimeAsString(char *timeStr);
void CVICALLBACK MyStopLog (int menuBar, int menuItem, void *callbackData,int
panel);
void CVICALLBACK MyStartLog (int menuBar, int menuItem, void *callbackData,int
panel);
static int Loging(void *intervals);



int Err;
static int panelConnect,panelMonitor;
static base_info* mainInfo;
static int log_lock,wg_lock,threadChecker,log_ch_lock,sensor_lock,logChecker;
static int WatchDogThread,LogingThread;
static int watchDogCounter;
int (__cdecl *ptrMyStopStreaming)(int, int, int, void *, int, int) =  MyStopStreaming;
int (__cdecl *ptrMyStartStreaming)(int, int, int, void *, int, int) =  MyStartStreaming;
static Task RetVal;
static char dataLog[MAX_LOG_LENGHT];
static double t_t;
static int threadID;

int main (int argc, char *argv[])
{
	if (InitCVIRTE (0, argv, 0) == 0)
		return -1;	/* out of memory */
	if ((panelConnect = LoadPanel (0, "Connection_Panel.uir", Connect)) < 0)
		return -1;
```

```
	CmtNewLock (NULL, 0, &wg_lock);
	CmtNewLock (NULL, 0, &log_lock);
	CmtNewLock (NULL, 0, &log_ch_lock);
	CmtNewLock (NULL, 0, &sensor_lock);
	DisplayPanel (panelConnect);
	panelMonitor=-1;
	RunUserInterface ();
	DiscardPanel (panelConnect);
	if(mainInfo!=NULL)
		Err = CloseCom (mainInfo->portnumber);



	free(mainInfo);
	return 0;
}

int CVICALLBACK QuitCallback (int panel, int control, int event,
                              void *callbackData, int eventData1,
int eventData2)
{
	switch (event)
	{
		case EVENT_COMMIT:
			if(panelMonitor!=-1)
			{
				freeCtrlSection(panelMonitor,mainInfo->sectionHead);
				CmtGetLock(sensor_lock);
				freelist(mainInfo->sensorHead);
				CmtReleaseLock(sensor_lock);
			}
			QuitUserInterface (0);
```

```
                        break;
        }
        return 0;
}

int CVICALLBACK MyConnect (int panel, int control, int event,
                                                void *callbackData, int eventData1, int
eventData2)
{
        switch (event)
        {
                case EVENT_COMMIT:
                        int portnumber;
                        GetCtrlVal (panel, Connect_Portnumber, &portnumber);
                                mainInfo=malloc(sizeof(base_info));
                        mainInfo->sensorHead=NULL;
                        mainInfo->sectionHead=malloc(sizeof(Section_Ctrl_Node));
                        mainInfo->portnumber=portnumber;

                        int Err = OpenComConfig (mainInfo->portnumber, "", Baud_Rate,
Parity, Data_Bits, Stop_Bits, SEND_BUFFER_SIZE, SEND_BUFFER_SIZE);
                        SetComTime (mainInfo->portnumber, 0.01);
                        FlushInQ (mainInfo->portnumber);
                        FlushOutQ (mainInfo->portnumber);
                        SendReqForSensors();
                        SetCtrlAttribute (panel,Connect_TIMER, ATTR_ENABLED, 1);

                        break;
        }
        return 0;
```

```
}

int CVICALLBACK MyOpenLog (int panel, int control, int event,
                                                    void *callbackData, int eventData1, int
eventData2)
{
        switch (event)
        {
                case EVENT_COMMIT:

                        break;
        }
        return 0;
}

int SendReqForSensors()
{
        char msg[2];
        msg[0]=Request_to_List_Sensors;
        msg[1]='\r';
        ComWrt (mainInfo->portnumber, msg, 2);
        StartWatchDog(waitForList);
        return 0;
}



int CVICALLBACK MyStartStreaming (int panel, int control, int event,
                                                        void *callbackData, int
eventData1, int eventData2)
{
```

```
	switch (event)
	{
		case EVENT_COMMIT:

			GetCtrlVal(panel,Monitor_Intervals,&t_t);
			char msg[20];
			msg[0]=Start_Data_Stream;
			memcpy(msg+1, &t_t, sizeof(double));
			msg[sizeof(double)+1]='\r';
			ComWrt (mainInfo->portnumber, msg, sizeof(double)+2);
			SetCtrlAttribute (panel, control,
ATTR_CALLBACK_FUNCTION_POINTER, ptrMyStopStreaming);
			SetCtrlAttribute (panel, control, ATTR_LABEL_TEXT, "Stop
streaming");
			StartWatchDog(waitForMassage);
			break;
	}
	return 0;
}

int CVICALLBACK MyStopStreaming (int panel, int control, int event,
                                 void *callbackData, int
eventData1, int eventData2)
{
	switch (event)
	{
		case EVENT_COMMIT:

			char msg[2];
			msg[0]=Stop_Data_Stream;
			msg[1]='\r';
			ComWrt (mainInfo->portnumber, msg, 2);
```

```
                    SetCtrlAttribute (panel, control,
ATTR_CALLBACK_FUNCTION_POINTER, ptrMyStartStreaming);
                    SetCtrlAttribute (panel, control, ATTR_LABEL_TEXT, "Start
streaming");
                    StopWatchDog();
                    break;
        }
        return 0;
}

void CVICALLBACK MyDisconnect (int menuBar, int menuItem, void *callbackData,int
panel)
{

        char status[100];
        SetCtrlAttribute (panel, Monitor_StartStreaming, ATTR_LABEL_TEXT, status);
        if(strcmp(status,"Stop streaming")==0)
        {

                char msg[2];
                msg[0]=Stop_Data_Stream;
                msg[1]='\r';
                ComWrt (mainInfo->portnumber, msg, 2);
                StopWatchDog();
        }

        if(panelMonitor!=-1)
        {
                freeCtrlSection(panelMonitor,mainInfo->sectionHead);
                CmtGetLock(sensor_lock);
                freelist(mainInfo->sensorHead);
                CmtReleaseLock(sensor_lock);
```

```
                DiscardPanel(panelMonitor);
                DisplayPanel(panelConnect);
                panelMonitor=-1;
        }

}

void CVICALLBACK MyRefresh (int menuBar, int menuItem, void *callbackData,
                                                        int panel)
{

        char status[100];
        GetCtrlAttribute (panel, Monitor_StartStreaming, ATTR_LABEL_TEXT, status);
        if(strcmp(status,"Stop streaming")==0)
        {
                char msg[2];
                msg[0]=Stop_Data_Stream;
                msg[1]='\r';
                ComWrt (mainInfo->portnumber, msg, 2);
                StopWatchDog();

        }
        StartWatchDog(waitForList);
        HidePanel(panelMonitor);
        freeCtrlSection(panelMonitor,mainInfo->sectionHead);
        CmtGetLock(sensor_lock);
        freelist(mainInfo->sensorHead);
        CmtReleaseLock(sensor_lock);
        mainInfo->sensorHead=NULL;
        mainInfo->sectionHead=malloc(sizeof(Section_Ctrl_Node));
        SendReqForSensors();
```

```
}


void CVICALLBACK MyMenuQuit (int menuBar, int menuItem, void *callbackData,
                                                        int panel)
{
        freeCtrlSection(panelMonitor,mainInfo->sectionHead);
        CmtGetLock(sensor_lock);
        freelist(mainInfo->sensorHead);
        CmtReleaseLock(sensor_lock);
        QuitUserInterface (0);
}

void CVICALLBACK MyExportLog (int menuBar, int menuItem, void *callbackData,
                                                        int panel)
{
        CmtGetLock(log_lock);
        CmtGetLock(log_ch_lock);
        if(logChecker)
        {
                if(!ConfirmPopup ("Warrning!", "To export the log data the loging process
needed to be stop\nare you sure you want to continue?"))
                {
                        CmtReleaseLock(log_lock);
                        CmtReleaseLock(log_ch_lock);
                        return;
                }
                logChecker=0;
                CmtReleaseLock(log_ch_lock);
```

```
		CmtWaitForThreadPoolFunctionCompletion
(DEFAULT_THREAD_POOL_HANDLE, LogingThread,
OPT_TP_PROCESS_EVENTS_WHILE_WAITING);

		SetMenuBarAttribute (menuBar,MENUBAR_Log_Start_Loging ,
ATTR_CALLBACK_FUNCTION_POINTER, MyStartLog);

		SetMenuBarAttribute (menuBar,MENUBAR_Log_Start_Loging ,
ATTR_ITEM_NAME, "Start loging");

	}

	else CmtReleaseLock(log_ch_lock);


	if(dataLog[0]=='\0')

	{

		MessagePopup ("Error", "Log data is empty cannot be export");

		CmtReleaseLock(log_lock);

		return;

	}

	char pathName[MAX_PATHNAME_LEN];

	if (FileSelectPopup("", "*.csv", "*.csv", "Save File", VAL_SAVE_BUTTON, 0, 0, 1, 1,
pathName))

  {

		FILE *fp = fopen(pathName, "w");

		fprintf(fp,"%s",dataLog);

		fclose(fp);

	}

	CmtReleaseLock(log_lock);


}

void CVICALLBACK MyStartLog (int menuBar, int menuItem, void *callbackData,int
panel)

{

	CmtGetLock(log_lock);

	if(dataLog[0]!='\0')
```

```
		if(!ConfirmPopup ("Warrning!", "There are already store log data if you start loging it will remove the old data are you sure you want to continue?"))
		{
			CmtReleaseLock(log_lock);
			return;
		}
	CmtReleaseLock(log_lock);
	double *t=malloc(sizeof(double));
	char *endptr;
	char answer[BUFFER_SIZE];
	PromptPopup ("Loging", "Please enter interal number", answer, BUFFER_SIZE-1);

	answer[strcspn(answer, " ,\n")] = 0;
	*t=strtod(answer, &endptr);
	CmtGetLock(log_ch_lock);
	logChecker=1;
	CmtReleaseLock(log_ch_lock);
	CmtScheduleThreadPoolFunction (DEFAULT_THREAD_POOL_HANDLE, Loging, (void *)t, &LogingThread);
	SetMenuBarAttribute (menuBar,menuItem , ATTR_CALLBACK_FUNCTION_POINTER, MyStopLog);
	SetMenuBarAttribute (menuBar,menuItem , ATTR_ITEM_NAME, "Stop loging");
}

void CVICALLBACK MyStopLog (int menuBar, int menuItem, void *callbackData,
                            int panel)
{
	CmtGetLock(log_lock);
	if(dataLog[0]!='\0')
		if(!ConfirmPopup ("Warrning!", "There are already store log data if you start loging it will remove the old data are you sure you want to continue?"))
		{
```

```
                    CmtReleaseLock(log_lock);
                    return;
                }
        CmtReleaseLock(log_lock);

        CmtGetLock(log_ch_lock);
        logChecker=0;
        CmtReleaseLock(log_ch_lock);
        CmtWaitForThreadPoolFunctionCompletion
(DEFAULT_THREAD_POOL_HANDLE, LogingThread,
OPT_TP_PROCESS_EVENTS_WHILE_WAITING);
        SetMenuBarAttribute (menuBar,menuItem ,
ATTR_CALLBACK_FUNCTION_POINTER, MyStartLog);
        SetMenuBarAttribute (menuBar,menuItem , ATTR_ITEM_NAME, "Start loging");
}

int CVICALLBACK MyTimer (int panel, int control, int event,
                                        void *callbackData, int eventData1, int
eventData2)
{
        switch (event)
        {
                case EVENT_TIMER_TICK:

                    AlertManager(RetVal);

                    char RecBuffer[SEND_BUFFER_SIZE];
                    unsigned char commade;
                    char *massage;
                    int l=ComRdTerm (mainInfo->portnumber, RecBuffer,
SEND_BUFFER_SIZE,'\r')-1;
                    commade=RecBuffer[0];
```

```
massage=RecBuffer+1;
switch (commade)
{
    case List_of_Sensors_Response:
        StopWatchDog();
        if(strlen(massage)>0)
        {
            UpdateSensorsSections(massage,l);
            if ((panelMonitor = LoadPanel (0,
"Monitor_panel.uir", Monitor)) < 0)
                return -1;
            HidePanel (panelConnect);
            CreateSections(panelMonitor,&mainInfo-
>sectionHead,mainInfo);
            DisplayPanel (panelMonitor);
        }else MessagePopup ("Error", "Error a connection
has been made but no sensor attach to the device");

        break;
    case Sensor_Data_Response:
            UpdateSensorsData(massage,l);

UpdateSectionData(panelMonitor,mainInfo,massage,NULL);
        break;
    case Data_Stream_Response:
            ResetWatchDog();
            UpdateSensorsData(massage,l);
            if(panelMonitor!=-1)
UpdateAllSectionData(panelMonitor,mainInfo);
        break;

}
```

```
                RecBuffer[0]=0;
                RecBuffer[1]=0;
                RecBuffer[2]='\r';

                break;
        }
        return 0;
}

void UpdateSensorsSections(char massage[],int l){
        int i=0;

        SensorType type;
        char id[MAX_SIZE_OF_ID];
        while(i<l)
        {
                int j=0;
                while(massage[i+j]!=0&&j<MAX_SIZE_OF_ID)
                {
                        id[j]=massage[i+j];
                        j++;
                }
                if(j==MAX_SIZE_OF_ID)return;
                i+=j;
                id[j]=massage[i++];
                type=(SensorType)massage[i++];
                CmtGetLock(sensor_lock);
                Node *sensorHead=addSensor(mainInfo->sensorHead,id,type);
                CmtReleaseLock(sensor_lock);
                if((mainInfo->sensorHead)==NULL)
                        mainInfo->sensorHead=sensorHead;
```

```
	}

	return;
}
void UpdateSensorsData(char massage[],int l){
		int i=0;

	SensorType type;
	char id[MAX_SIZE_OF_ID];
	while(i<l)
	{
		int j=0;
		while(massage[i+j]!=0)
		{
			id[j]=massage[i+j];
			j++;
		}
		i+=j;
		id[j]=massage[i++];
		double data=*((double *)(massage+i));
		i+=sizeof(double);
		CmtGetLock(sensor_lock);
		setSensorData(mainInfo->sensorHead,id,data);
		CmtReleaseLock(sensor_lock);
	}
}
void StartWatchDog(Task task)
{
	Task *alert=malloc(sizeof(Task));
	*alert=task;
```

```
	CmtGetLock (wg_lock);
	threadChecker=1;
	CmtReleaseLock (wg_lock);
	CmtScheduleThreadPoolFunction (DEFAULT_THREAD_POOL_HANDLE,
WatchDog, ((void*)alert), &WatchDogThread);
}
void StopWatchDog()
{
	CmtGetLock (wg_lock);
	threadChecker=-1;
	CmtReleaseLock (wg_lock);



}
void ResetWatchDog()
{
	CmtGetLock (wg_lock);
	threadChecker=-2;
	CmtReleaseLock (wg_lock);


}

static int WatchDog(void *task)
{
	int maxcount =10;
	Task tmpA=*(Task*)task;
	Task alert=tmpA;
	free(task);
	double t=(5000*1e-3)/maxcount;
	CmtGetLock (wg_lock);
	watchDogCounter=0;
```

```
		while(threadChecker==1)
		{
			CmtReleaseLock (wg_lock);
			watchDogCounter++;
			Delay(t);
			CmtGetLock (wg_lock);
			if(watchDogCounter>=maxcount&&threadChecker==1)
				threadChecker=0;
			if(threadChecker==-2)
			{
				watchDogCounter=0;
				threadChecker=1;
			}
		}


		if(threadChecker!=0)
			alert=0;
		RetVal=alert;
		CmtReleaseLock (wg_lock);
		CmtExitThreadPoolThread (0);
		return 0;

}

void AlertManager(Task task)
{
		int alertMassage;
		char msg[20];
		switch (task)
		{
```

```
		case waitForList:


			MessagePopup ("Alert", "Time out\nNo respond from device");
			StopWatchDog();
			if(panelMonitor!=-1)
			{
				DiscardPanel(panelMonitor);
				panelMonitor=-1;
			}
			DisplayPanel(panelConnect);
			break;
		case waitForMassage:
			int select = GenericMessagePopup ("Alert", "Time out\nNo respond from device!\nWould you like to:", "Exit Program", "Disconnect", "keep waiting", NULL,0 , 0, VAL_GENERIC_POPUP_BTN1, VAL_GENERIC_POPUP_BTN1, VAL_GENERIC_POPUP_BTN3);


			switch (select)
			{
				case VAL_GENERIC_POPUP_BTN1:
					QuitUserInterface (0);
					break;
				case VAL_GENERIC_POPUP_BTN2:
					msg[0]=Stop_Data_Stream;
					msg[1]='\r';
					ComWrt (mainInfo->portnumber, msg, 2);
					freeCtrlSection(panelMonitor,mainInfo->sectionHead);
					CmtGetLock(sensor_lock);
					freelist(mainInfo->sensorHead);
					CmtReleaseLock(sensor_lock);
					DiscardPanel(panelMonitor);
```

```
                                        panelMonitor=-1;
                                        DisplayPanel(panelConnect);
                                        break;
                                case VAL_GENERIC_POPUP_BTN3:
                                        StartWatchDog(waitForMassage);
                                        msg[0]=Start_Data_Stream;
                                        memcpy(msg+1, &t_t, sizeof(double));
                                        msg[sizeof(double)+1]='\r';
                                        ComWrt (mainInfo->portnumber, msg,
sizeof(double)+2);
                                        break;
                        }
                        break;

        }
        RetVal=0;

}


void GetCurrentTimeAsString(char *timeStr) {
   time_t currentTime;
   struct tm *localTime;
   time(&currentTime);

   localTime = localtime(&currentTime);

   strftime(timeStr, 50, "%c", localTime);
}

static int Loging(void *intervals)
```

```
{
	double t=*(double*)intervals;
	char tmpBuff[MAX_LOG_LENGHT];
	CmtGetLock(log_lock);
	sprintf(dataLog,"Time");

	CmtGetLock(sensor_lock);
	Node *tmp=mainInfo->sensorHead;
	while(tmp!=NULL)
	{
		char* type=getSensorTypeString(getSensorType(mainInfo->sensorHead,tmp->sensor->id));
		sprintf(tmpBuff,"%s,%s Type: %s",dataLog,tmp->sensor->id,type);
		strcpy(dataLog,tmpBuff);
		tmp=tmp->next;
	}
	sprintf(tmpBuff,"%s\n",dataLog);
	strcpy(dataLog,tmpBuff);

	CmtReleaseLock(sensor_lock);
	CmtReleaseLock(log_lock);


	CmtGetLock(log_ch_lock);
	while(logChecker)
	{
		CmtReleaseLock(log_ch_lock);
		CmtGetLock(sensor_lock);
		CmtGetLock(log_lock);
		tmp=mainInfo->sensorHead;
		char timestr[BUFFER_SIZE];
```

```
                GetCurrentTimeAsString(timestr);
                sprintf(tmpBuff,"%s%s",dataLog,timestr);
                strcpy(dataLog,tmpBuff);
                while(tmp!=NULL)
                {
                        sprintf(tmpBuff,"%s,%.3f",dataLog,tmp->sensor->data);
                        strcpy(dataLog,tmpBuff);
                        tmp=tmp->next;
                }
                CmtReleaseLock(sensor_lock);
                sprintf(tmpBuff,"%s\n",dataLog);
                strcpy(dataLog,tmpBuff);
                CmtReleaseLock(log_lock);

                Delay(t);
                CmtGetLock(log_ch_lock);
        }
        CmtReleaseLock(log_ch_lock);
        free(intervals);
        CmtExitThreadPoolThread (0);


}
```

6.7. emulator.c

```
#include <utility.h>
#include <rs232.h>
#include <ansi_c.h>
#include <cvirte.h>
#include <time.h>
#include <userint.h>
#include "emulator.h"
#include "sensors.h"
#include "communication.h"

#define MAX_SECTIONS 11
#define DISTANT_BETWEEN_SECTION 40
#define DISTANT_BETWEEN_CTRL 90
#define DISTANT_FROM_LEFT 10
#define DISTANT_FROM_TOP 50
#define MAX_LOG_SIZE 10000000
#define BUFFER_SIZE 200

typedef enum{
	offline,
	idle,
	sending_list,  //stata for sending list of connected sensors
	sending_data,		//stata for sending singel sensor data
	streaming, //stata for streaming data accourding to the inteval
} state;

//////////////////////Define Structs
typedef struct {
	int inx;
```

```
        char id_sensor[MAX_SIZE_OF_ID];
  int id_ctrl;
        int type_ctrl;
  int data_ctrl;
  int updateButton_ctrl;
  int removeButton_ctrl;
} Section;

typedef struct SectionNode{
        Section *section;
  struct SectionNode *next;
} SectionNode;



////////////////////////////////



/////////Define global parametes
static int panelHandle,number_of_section;
static int panelMonitorHandle=-1;
static Node* sensorHead;
static SectionNode* sectionHead;
static int ComPort=4;
int Err;
static int streaming_on;
static state currect_state=offline;
static int sensor_lock,sm_lock,log_lock;
static int threadFunctionId,rdThread;
static char sendLog[MAX_LOG_SIZE];
static char recieveLog[MAX_LOG_SIZE];
```

```
static int pollingMode =1;

static int rd_on=0;

////////////////////////////////


void insertToLog(char source[],char newMassage[]);

void CVICALLBACK RecieveCallback(int portNumber, int eventMask, void
*callbackData);

Section* initializeNewSection(int panel,char *id,int inx);

void RemoveSectionNode(int panel,SectionNode **sectionHead,Node **sensorhead,
int index);

SectionNode* addSection(int panel,SectionNode *sectionHead, char *id,int inx);

int FindSectionIndexByUpdateButton(SectionNode *head, int updateButton_ctrl);

int FindSectionIndexByRemoveButton(SectionNode *head, int removeButton_ctrl);

int UpdateSensorFromSectionByInx(int panel, int inx,SectionNode *sectionhead,Node
*sensorHead);

int CVICALLBACK MyRemove (int panel, int control, int event,void *callbackData, int
eventData1, int eventData2);

int CVICALLBACK MyUpdate (int panel, int control, int event,void *callbackData, int
eventData1, int eventData2);

int CVICALLBACK MyDisconnect (int panel, int control, int event,void *callbackData, int
eventData1, int eventData2);

int CVICALLBACK MyConnect (int panel, int control, int event,void *callbackData, int
eventData1, int eventData2);

static int streamingRun(void * intervales) ;

void smControl(state next_state,char massage[]) ;

void sendList();

void sendData(char massage[]);

void GetCurrentTimeAsString(char *timeStr);

static int RandChange(void *a);

void UpdateSectionData(SectionNode *sectionHead,int inx,double data) ;


int (__cdecl *ptrMyRemove)(int, int, int, void *, int, int) =  MyRemove;

int (__cdecl *ptrMyUpdate)(int, int, int, void *, int, int) =  MyUpdate;
```

```
int (__cdecl *ptrMyConnect)(int, int, int, void *, int, int) =  MyConnect;
int (__cdecl *ptrMyDisconnect)(int, int, int, void *, int, int) =  MyDisconnect;



////////////////////////////////////////////////////////
//
//                  All struct function for creating a section and managing it
//
////////////////////////////////////////////////////////


Section* initializeNewSection(int panel,char *id,int inx) {
        if(inx>MAX_SECTIONS) return NULL;
        Section* newSection=malloc(sizeof(Section));
        if (newSection == NULL) {
    return NULL;
  }
  strncpy(newSection->id_sensor, id, MAX_SIZE_OF_ID*sizeof(char));
  newSection->inx = inx;
  // Create controls dynamically
  newSection->id_ctrl = NewCtrl(panel, CTRL_STRING, "ID", inx *
DISTANT_BETWEEN_SECTION+DISTANT_FROM_TOP, DISTANT_FROM_LEFT);
  newSection->type_ctrl = NewCtrl(panel, CTRL_STRING, "Type", inx *
DISTANT_BETWEEN_SECTION+DISTANT_FROM_TOP,
DISTANT_FROM_LEFT+DISTANT_BETWEEN_CTRL);
        newSection->data_ctrl = NewCtrl(panel, CTRL_NUMERIC, "data", inx *
DISTANT_BETWEEN_SECTION+DISTANT_FROM_TOP,
DISTANT_FROM_LEFT+DISTANT_BETWEEN_CTRL*2);
  newSection->updateButton_ctrl = NewCtrl(panel,
CTRL_SQUARE_COMMAND_BUTTON, "Update Value", inx *
DISTANT_BETWEEN_SECTION+DISTANT_FROM_TOP,
DISTANT_FROM_LEFT+DISTANT_BETWEEN_CTRL*3);
```

```
   newSection->removeButton_ctrl = NewCtrl(panel,
CTRL_SQUARE_COMMAND_BUTTON, "Remove", inx *
DISTANT_BETWEEN_SECTION+DISTANT_FROM_TOP,
DISTANT_FROM_LEFT+DISTANT_BETWEEN_CTRL*4);


        //set control mode and other setting for each ctrl
        SetCtrlAttribute (panel,newSection->id_ctrl , ATTR_CTRL_MODE, 0);
        SetCtrlAttribute (panel,newSection->type_ctrl , ATTR_CTRL_MODE, 0);
        SensorType tmptype=getSensorType(sensorHead,id);
        char* type_str=getSensorTypeString(tmptype);
        SetCtrlVal (panel, newSection->type_ctrl, type_str);
        SetCtrlVal (panel, newSection->id_ctrl, id);
        SetCtrlAttribute (panel, newSection->updateButton_ctrl,
ATTR_CALLBACK_FUNCTION_POINTER, ptrMyUpdate);
        SetCtrlAttribute (panel, newSection->removeButton_ctrl,
ATTR_CALLBACK_FUNCTION_POINTER, ptrMyRemove);


        return newSection;
}


void RemoveSectionNode(int panel,SectionNode **sectionHead,Node **sensorhead,
int index) {
   SectionNode *current = *sectionHead;
   SectionNode *previous = NULL;


   // Find the node to remove
   while (current != NULL && current->section->inx != index) {
      previous = current;
      current = current->next;
   }


   // If the node was not found, return
   if (current == NULL) {
```

```
    return;
  }

  // Remove the node
  if (previous == NULL) {
    // If the node to remove is the first node, update the head pointer
    *sectionHead = current->next;
  } else {
    // Otherwise, update the previous node's next pointer
    previous->next = current->next;
  }

  //Discard the ctrls
        DiscardCtrl(panel,current->section->id_ctrl);
  DiscardCtrl(panel, current->section->type_ctrl);
  DiscardCtrl(panel, current->section->data_ctrl);
  DiscardCtrl(panel, current->section->updateButton_ctrl);
        DiscardCtrl(panel, current->section->removeButton_ctrl);
        CmtGetLock (sensor_lock);
        removeSensor(sensorhead,current->section->id_sensor);
        CmtReleaseLock (sensor_lock);

        // Free the memory allocated for the node
  free(current->section);
  free(current);
        current=NULL;

  // Update the inx field of the remaining sections
  current = *sectionHead;
  int newIndex = 0;
  while (current != NULL) {
```

```
    current->section->inx = newIndex;
    newIndex++;
    current = current->next;
  }
        current=*sectionHead;
        while(current!=NULL)
        {
    SetCtrlAttribute(panel, current->section->id_ctrl, ATTR_TOP, ((current->section->inx) * DISTANT_BETWEEN_SECTION)+DISTANT_FROM_TOP);
    SetCtrlAttribute(panel, current->section->type_ctrl, ATTR_TOP, ((current->section->inx) * DISTANT_BETWEEN_SECTION)+DISTANT_FROM_TOP);
    SetCtrlAttribute(panel, current->section->data_ctrl, ATTR_TOP, ((current->section->inx) * DISTANT_BETWEEN_SECTION)+DISTANT_FROM_TOP);
    SetCtrlAttribute(panel, current->section->updateButton_ctrl, ATTR_TOP, ((current->section->inx) * DISTANT_BETWEEN_SECTION)+DISTANT_FROM_TOP);
                SetCtrlAttribute(panel, current->section->removeButton_ctrl, ATTR_TOP, ((current->section->inx) * DISTANT_BETWEEN_SECTION)+DISTANT_FROM_TOP);
                current=current->next;
        }

}

SectionNode* addSection(int panel,SectionNode *sectionHead, char *id,int inx) {
        SectionNode *newNodeSection = (SectionNode*) malloc(sizeof(SectionNode));

        if(newNodeSection==NULL) return NULL; //Cannot allocate memory for new new Node Section
        newNodeSection->section=initializeNewSection(panel,id, inx);
        if(newNodeSection->section==NULL) return NULL; //Cannot allocate memory for new Section
        newNodeSection->next=NULL;
        if(sectionHead==NULL) sectionHead=newNodeSection;
        else
```

```
        {
                SectionNode *tmpNode=sectionHead;
                while(tmpNode->next!=NULL) tmpNode=tmpNode->next;
                tmpNode->next=newNodeSection;

        }
  return sectionHead;
}

int FindSectionIndexByUpdateButton(SectionNode *head, int updateButton_ctrl) {
  SectionNode *current = head;
  while (current != NULL) {
    if (current->section->updateButton_ctrl == updateButton_ctrl) {
      return current->section->inx;
    }
    current = current->next;
  }
  return -1; // Indicate that the section was not found
}

int UpdateSensorFromSectionByInx(int panel, int inx,SectionNode *sectionhead,Node
*sensorHead) {
  SectionNode *current = sectionhead;
  while ((current != NULL)&&current->section->inx != inx) {
    current = current->next;
  }
        if(current==NULL) return -1; // Indicate that the section was not found
        char tmpId[MAX_SIZE_OF_ID];
        strcpy(tmpId,current->section->id_sensor);
        double data;
        GetCtrlVal (panel,current->section->data_ctrl , &data);
```

```
		CmtGetLock (sensor_lock);
		setSensorData(sensorHead,tmpId,data);
		CmtReleaseLock (sensor_lock);

		return 0;
}

int FindSectionIndexByRemoveButton(SectionNode *head, int removeButton_ctrl) {
   SectionNode *current = head;
   while (current != NULL) {
      if (current->section->removeButton_ctrl == removeButton_ctrl) {
         return current->section->inx;
      }
      current = current->next;
   }
   return -1; // Indicate that the section was not found
}

void freeSections(int panel,SectionNode *head)
{
		if(head!=NULL)
		{
			freeSections(panel,head->next);
			DiscardCtrl(panel,head->section->id_ctrl);
		DiscardCtrl(panel, head->section->type_ctrl);
		DiscardCtrl(panel, head->section->data_ctrl);
		DiscardCtrl(panel, head->section->updateButton_ctrl);
			DiscardCtrl(panel, head->section->removeButton_ctrl);
			free(head->section);
			free(head);
		}else number_of_section =0;
```

```
}

////////////////////////////////////////////////////////////////////////////////////////////////

int main (int argc, char *argv[])
{
	if (InitCVIRTE (0, argv, 0) == 0)
		return -1;	/* out of memory */
	if ((panelHandle = LoadPanel (0, "emulator.uir", PANEL)) < 0)
		return -1;

	int i;
	for(i=0;i<NUMBER_OF_SENSORS_TYPES;i++)
	{
		InsertListItem (panelHandle, PANEL_Sensor_type, i,
getSensorTypeString(i), i);
	}
	CmtNewLock (NULL, 0, &sensor_lock);
	CmtNewLock (NULL, 0, &sm_lock);
	CmtNewLock (NULL, 0, &log_lock);
	sensorHead=NULL;
	sectionHead=NULL;
	DisplayPanel (panelHandle);
	RunUserInterface ();
	freeSections(panelHandle,sectionHead);
	DiscardPanel (panelHandle);

}
```

```
	int Err = CloseCom (ComPort);
	freelist(sensorHead);

	return 0;
}


int CVICALLBACK QuitCallback (int panel, int control, int event,
                              void *callbackData, int eventData1,
int eventData2)
{
	switch (event)
	{
		case EVENT_COMMIT:
			QuitUserInterface (0);
			break;
	}
	return 0;
}

int CVICALLBACK add_new_Sensor (int panel, int control, int event,
                                void *callbackData, int eventData1, int
eventData2)
{
	switch (event)
	{
		case EVENT_COMMIT:
			char tmpId[MAX_SIZE_OF_ID];
			int tmp;
			SensorType tmpType;
			GetCtrlVal (panel, PANEL_Set_ID, tmpId);
```

```
                        GetCtrlVal (panel, PANEL_Sensor_type, &tmp);
                        if((findSensor(sensorHead,
tmpId)==NULL)&&number_of_section<MAX_SECTIONS&&tmpId[0]!=0)
                        {
                                tmpType=(SensorType)tmp;
                                sensorHead=addSensor(sensorHead,tmpId,tmpType);

        sectionHead=addSection(panel,sectionHead,tmpId,number_of_section);
                                number_of_section++;
                        }

                        break;
        }
        return 0;
}


int CVICALLBACK MyRemove (int panel, int control, int event,
                                        void *callbackData, int eventData1, int
eventData2)
{
        switch (event)
        {
                case EVENT_COMMIT:
                        int in=FindSectionIndexByRemoveButton(sectionHead,control);
                        RemoveSectionNode(panel,&sectionHead,&sensorHead,in);
                        number_of_section--;
                        break;
        }
        return 0;
}
```

```
int CVICALLBACK MyUpdate (int panel, int control, int event,
                                                    void *callbackData, int eventData1, int
eventData2)
{
        switch (event)
        {
                case EVENT_COMMIT:

                        int inx=FindSectionIndexByUpdateButton(sectionHead,control);
                        UpdateSensorFromSectionByInx(panel,
inx,sectionHead,sensorHead);

                        break;
        }
        return 0;
}



void CVICALLBACK SaveTemplate (int menuBar, int menuItem, void *callbackData,
                                                        int panel)
{
        if(sectionHead==NULL)
        {
                MessagePopup("Error", "Sensor list is empty!");
                return;
        }
        char pathName[MAX_PATHNAME_LEN];
        if (FileSelectPopup("", "*.txt", "*.txt", "Save File", VAL_SAVE_BUTTON, 0, 0, 1, 1,
pathName))
  {
                FILE *fp = fopen(pathName, "w");
```

```
                Node* tmp=sensorHead;
                char *id;
                double data;
                SensorType type;
                while(tmp!=NULL)
                {
                        id=tmp->sensor->id;
                        type=tmp->sensor->type;
                        data=tmp->sensor->data;
                        fprintf(fp,"%s ,%d,% .3f\n",id,type,data);

                        tmp=tmp->next;
                }
                fclose(fp);
        }

}

void CVICALLBACK LoadTemplate (int menuBar, int menuItem, void *callbackData,
                                                        int panel)
{
        CmtGetLock(sm_lock);
        if(currect_state==streaming)
        {
                MessagePopup("Error!", "The emolator is streaming data this oparetion
cannot be complate at this time.");
                CmtReleaseLock(sm_lock);
                return;
        }
        CmtReleaseLock(sm_lock);
        if(sectionHead!=NULL)
```

```
	{
		int pick=ConfirmPopup ("Warrning!", "This action will remove all the existing sensors\nWould you still like to contiunue?");
		if(pick==0)
			return;
	}
	freeSections(panel,sectionHead);
	freelist(sensorHead);
	sectionHead=NULL;
	sensorHead=NULL;
	char pathName[MAX_PATHNAME_LEN];
	if (FileSelectPopup("", "*.txt", "*.txt", "Load File", VAL_LOAD_BUTTON, 0, 0, 1, 1, pathName))
  {
		FILE *fp = fopen(pathName, "r");
		char *id;
		char *dataStr;
		double data;
		SensorType type;
		char *typeStr;
		char *endptr;
		char buffer[BUFFER_SIZE];
		number_of_section=0;
		while(fgets(buffer, BUFFER_SIZE, fp))
		{


			id = strtok(buffer, ",");
			id[strcspn(id, " ,\n")] = 0;
```

```
                        typeStr=strtok(NULL, ",");
                        typeStr[strcspn(typeStr, " ,\n")] = 0;
                        type=(SensorType)atoi(typeStr);

                        dataStr = strtok(NULL, ",");
                        //dataStr[strcspn(dataStr, " ,\n")] = 0;
                        data= strtod(dataStr, &endptr);

                        if((findSensor(sensorHead,
id)==NULL)&&number_of_section<MAX_SECTIONS&&id[0]!=0)
                        {
                                sensorHead=addSensor(sensorHead,id,type);

        sectionHead=addSection(panelHandle,sectionHead,id,number_of_section);

        UpdateSectionData(sectionHead,number_of_section,data);
                                UpdateSensorFromSectionByInx(panel,
number_of_section,sectionHead,sensorHead);

                                number_of_section++;
                        }

                }
                fclose(fp);
        }


}

void CVICALLBACK menu_exit (int menuBar, int menuItem, void *callbackData,
                                                        int panel)
{
```

```
	QuitUserInterface (0);
}

void CVICALLBACK OpenLog (int menuBar, int menuItem, void *callbackData,
                                                    int panel)
{
	if(panelMonitorHandle!=-1) return;



	if ((panelMonitorHandle = LoadPanel (0, "emulator.uir", MONITOR)) < 0)
		return;
		DisplayPanel (panelMonitorHandle);
		ResetTextBox (panelMonitorHandle,MONITOR_SendMSG , "");
		ResetTextBox (panelMonitorHandle,MONITOR_RecieveMSG , "");
		InsertTextBoxLine (panelMonitorHandle, MONITOR_SendMSG, 0,
sendLog);
		InsertTextBoxLine (panelMonitorHandle, MONITOR_RecieveMSG, 0,
recieveLog);



}


void WriteLogToFile(const char* log, const char* filename) {


	if(log[0]=='\0')
	{
		MessagePopup("Error", "Log data in empty");
		return;
	}
	char pathName[MAX_PATHNAME_LEN];
```

```
        char title[MAX_PATHNAME_LEN];
        sprintf(title,"Save %s file",filename);
        if (FileSelectPopup("", "*.txt", "*.txt", title, VAL_SAVE_BUTTON, 0, 0, 1, 1,
pathName))
        {
          FILE *file = fopen(pathName, "w");
          if (file != NULL) {
            for (int i = 0; log[i] != '\0' && i < MAX_LOG_SIZE; ++i) {
              fputc(log[i], file);
            }
            fclose(file);
          } else {
            MessagePopup("Error", "File opening failed");
          }
        }
}

void CVICALLBACK Save_command_log (int menuBar, int menuItem, void
*callbackData,
                                                    int panel)
{

        CmtGetLock(log_lock);
        WriteLogToFile(sendLog, "send log");
   WriteLogToFile(recieveLog, "recieve log");
        CmtReleaseLock(log_lock);

}

int CVICALLBACK MyConnect (int panel, int control, int event,
                                          void *callbackData, int eventData1, int
eventData2)
```

```
{
	switch (event)
	{
		case EVENT_COMMIT:
			GetCtrlVal (panel, PANEL_Com_Port, &ComPort);
			int Err = OpenComConfig (ComPort, "", Baud_Rate, Parity,
Data_Bits, Stop_Bits, SEND_BUFFER_SIZE, SEND_BUFFER_SIZE);
			SetComTime (ComPort, 0.01);
			FlushInQ (ComPort);
			FlushOutQ (ComPort);
			if(pollingMode)
			{
				SetCtrlAttribute (panel, PANEL_TIMER,
ATTR_ENABLED, 1);

			}
			else
			{
				InstallComCallback (ComPort, LWRS_RXFLAG, 0, 0,
RecieveCallback, 0);
			}

			SetCtrlAttribute (panel, control,
ATTR_CALLBACK_FUNCTION_POINTER, ptrMyDisconnect);
			SetCtrlAttribute (panel, control, ATTR_LABEL_TEXT, "Disconnect");
			smControl(idle,NULL);


			break;
	}
	return 0;
}
```

```
int CVICALLBACK MyDisconnect (int panel, int control, int event,
                                                    void *callbackData, int eventData1, int
eventData2)
{
        switch (event)
        {
                case EVENT_COMMIT:
                        smControl(offline,NULL);
                        int Err = CloseCom (ComPort);
                        SetCtrlAttribute (panel, control,
ATTR_CALLBACK_FUNCTION_POINTER, ptrMyConnect);
                        SetCtrlAttribute (panel, control, ATTR_LABEL_TEXT, "Connect");
                        if(pollingMode)
                        {
                                        SetCtrlAttribute (panel, PANEL_TIMER,
ATTR_ENABLED, 0);

                        }


                        break;
        }
        return 0;
}

void CVICALLBACK RecieveCallback(int portNumber, int eventMask, void
*callbackData)
{

        char RecChar[2];
        char massage[SEND_BUFFER_SIZE];
```

```
		ComRd (ComPort, RecChar, 1);
		char commade=RecChar[0];

		ComRd (ComPort, massage, SEND_BUFFER_SIZE-1);
		switch(commade)
		{
			case Request_to_List_Sensors:
				smControl(sending_list,NULL);
				smControl(idle,NULL);
			break;
			case Request_Sensor_Data:
				smControl(sending_data,massage);
				smControl(idle,NULL);
			break;
			case Start_Data_Stream:
				smControl(streaming,massage);
			break;
			case Stop_Data_Stream:
				smControl(idle,NULL);
			break;
			default: return ;
		}
		char tmp[100];
		sprintf(tmp,"%c%s" ,commade,massage);
		insertToLog(recieveLog,tmp);

		return ;
}
void insertToLog(char source[],char newMassage[])
{
```

```
		char currentTimeStr[50];
		GetCurrentTimeAsString(currentTimeStr);
		sprintf(source,  "%s%s> %s\n",source,currentTimeStr,newMassage);
		if(panelMonitorHandle!=-1)
		{
			ResetTextBox (panelMonitorHandle,MONITOR_SendMSG , "");
			InsertTextBoxLine (panelMonitorHandle, MONITOR_SendMSG, 0,
sendLog);
			ResetTextBox (panelMonitorHandle,MONITOR_RecieveMSG , "");
			InsertTextBoxLine (panelMonitorHandle, MONITOR_RecieveMSG, 0,
recieveLog);
		}
}

void CVICALLBACK Close_window (int menuBar, int menuItem, void *callbackData,
                                                        int panel)
{
		if(panel!=-1)
		{
			DiscardPanel (panel);
			if(panel==panelMonitorHandle)
			{
				panelMonitorHandle=-1;
				return;
			}
		}

}
void smControl(state next_state,char massage[])
{
		switch(currect_state)
```

```
        {
                case offline:
                        if (next_state==idle) currect_state=idle;
                break;
                case idle:
                        if(next_state==sending_list)
                        {
                                currect_state=sending_list;
                                sendList();
                        }
                        if(next_state==sending_data)
                        {
                                currect_state=sending_data;
                                sendData(massage);
                        }
                        if(next_state==streaming)
                        {
                                CmtGetLock (sm_lock);
                                streaming_on=1;
                                CmtReleaseLock (sm_lock);
                                CmtScheduleThreadPoolFunction
(DEFAULT_THREAD_POOL_HANDLE, streamingRun, ((double *)massage),
&threadFunctionId);
                                currect_state=streaming;
                        }
                        if(next_state==offline) currect_state=offline;
                break;
                case sending_list:
                        if (next_state==idle) currect_state=idle;
                        if (next_state==offline) currect_state=offline;
                break;
```

```
                case sending_data:
                        if (next_state==idle) currect_state=idle;
                        if (next_state==offline) currect_state=offline;
                break;
                case streaming:

                        if (next_state==idle||next_state==offline)
                        {
                                CmtGetLock (sm_lock);
                                streaming_on=0;
                                CmtReleaseLock (sm_lock);
                                CmtWaitForThreadPoolFunctionCompletion
(DEFAULT_THREAD_POOL_HANDLE, threadFunctionId,
OPT_TP_PROCESS_EVENTS_WHILE_WAITING);
                                CmtReleaseThreadPoolFunctionID
(DEFAULT_THREAD_POOL_HANDLE, threadFunctionId);
                                currect_state=next_state;
                        }
                break;



        }
}

void sendData(char massage[]){
        double data=getSensorData(sensorHead,massage);
        char buffer[SEND_BUFFER_SIZE];
        char tmplog[SEND_BUFFER_SIZE];
        tmplog[0]=0;
        sprintf(tmplog,"0xA2%s%.2f",massage,data);
        int i=1;
        sprintf(buffer,"%c%s",Sensor_Data_Response,massage);
```

```
		i+=strlen(buffer);
		memcpy(buffer+i, &data, sizeof(double));
		i+=sizeof(double);
		insertToLog(sendLog,tmplog);
		buffer[i++]='\r';
		ComWrt (ComPort, buffer, i);
}



void sendList()
{
		char buffer[SEND_BUFFER_SIZE];
		char tmplog[SEND_BUFFER_SIZE];
		char tmplog2[SEND_BUFFER_SIZE];
		buffer[0]=List_of_Sensors_Response;
		int i=1;
		Node* tmp = sensorHead;
		CmtGetLock(sensor_lock);
		while(tmp!=NULL&&i<(SEND_BUFFER_SIZE-1))
		{
			int j=0;

			while(tmp->sensor-
>id[j]!=0&&j<MAX_SIZE_OF_ID&&(i+j)<(SEND_BUFFER_SIZE-1))
			{
			buffer[i+j]=tmp->sensor->id[j];

				j++;
			}
			i+=j;
```

```
                sprintf(tmplog2,"%s%s%s",tmplog,tmp->sensor->id,getSensorTypeString(tmp->sensor->type));
                buffer[i++]=0;
                strcpy(tmplog,tmplog2);
                if(i<(SEND_BUFFER_SIZE-1)) buffer[i++]=(char)tmp->sensor->type;
                tmp=tmp->next;;
        }
        CmtReleaseLock(sensor_lock);
        if(i>=(SEND_BUFFER_SIZE))
        {
                buffer[SEND_BUFFER_SIZE-1]='\r';
                ComWrt (ComPort, buffer, SEND_BUFFER_SIZE);
        }
        else
        {
                buffer[i++]='\r';
                ComWrt (ComPort, buffer, i);
        }
        sprintf(tmplog2,"0xA1%s",tmplog);
        insertToLog(sendLog,tmplog2);
        strcpy(tmplog2,"");
        strcpy(tmplog,"");


}

static int streamingRun(void *intervales)
{
        double t=0;
        memcpy(&t,intervales,sizeof(double));
        CmtGetLock (sm_lock);
```

```
int a=streaming_on;
CmtReleaseLock (sm_lock);
char buffer[SEND_BUFFER_SIZE];
char tmplog[SEND_BUFFER_SIZE];
char tmplog2[SEND_BUFFER_SIZE];
buffer[0]=Data_Stream_Response;

while(a)
{
	buffer[0]=Data_Stream_Response;
	int i=1;
	int k=0;
	Node* tmp = sensorHead;
	while(tmp!=NULL&&i<(SEND_BUFFER_SIZE-1))
	{

		int j=0;
		while(tmp->sensor->id[j]!=0&&j<MAX_SIZE_OF_ID&&(i+j)<SEND_BUFFER_SIZE&&(i+j)<(SEND_BUFFER_SIZE-1))
		{
			buffer[i+j]=tmp->sensor->id[j];
			j++;
		}
		i+=j;

		if(i>=(SEND_BUFFER_SIZE)) break;
		tmplog[k+j]=0;
		buffer[i++]=0;

		if((i+sizeof(double))<(SEND_BUFFER_SIZE-1))
```

```
                {

                        CmtGetLock (sensor_lock);
                        double data =tmp->sensor->data;
                        CmtReleaseLock (sensor_lock);

                        memcpy(buffer+i, &data, sizeof(double));
                        sprintf(tmplog2,"%s%s;%.2f",tmplog,tmp->sensor-
>id,data);
                        k+=strlen(tmplog2);
                        strcpy(tmplog,tmplog2);
                        strcpy(tmplog2,"");
                        i+=sizeof(double);

                }

                tmplog[k]=0;
                tmp=tmp->next;
        }
        if((i+sizeof(double))>=(SEND_BUFFER_SIZE))
        {
                tmplog[SEND_BUFFER_SIZE-1]='\r';
                buffer[SEND_BUFFER_SIZE-1]='\r';
                ComWrt (ComPort, buffer, SEND_BUFFER_SIZE);
        }
        else
        {
                tmplog[i-1]='\r';
                buffer[i++]='\r';
                ComWrt (ComPort, buffer, i);
        }
```

```
                tmplog[k]=0;
                sprintf(tmplog2,"0xA3%s",tmplog);
                CmtGetLock(log_lock);
                insertToLog(sendLog,tmplog2);
                CmtReleaseLock(log_lock);
                strcpy(tmplog,"");
                strcpy(tmplog2,"");
                Delay(t);

                CmtGetLock (sm_lock);
                a=streaming_on;
                CmtReleaseLock (sm_lock);
        }

        return 0;
}



int CVICALLBACK MyTimer (int panel, int control, int event,
                                        void *callbackData, int eventData1, int
eventData2)
{
        switch (event)
        {
                case EVENT_TIMER_TICK:


                        char RecChar[SEND_BUFFER_SIZE];
                        unsigned char commade;
```

```
char *massage;
ComRdTerm (ComPort, RecChar, SEND_BUFFER_SIZE,'\r');
commade=RecChar[0];
massage=RecChar+1;
char tmp[100];
switch(commade)
{
	case Request_to_List_Sensors:
		sprintf(tmp,"0x01");
		smControl(sending_list,NULL);
		smControl(idle,NULL);
	break;
	case Request_Sensor_Data:
		sprintf(tmp,"0x02%s", massage);
		smControl(sending_data,massage);
		smControl(idle,NULL);
	break;
	case Start_Data_Stream:

		sprintf(tmp,"0x03%.2f", ((double *)massage)[0]);
		smControl(streaming,massage);
	break;
	case Stop_Data_Stream:
		sprintf(tmp,"0x04%s", massage);
		smControl(idle,NULL);
	break;
	default: return 0;
}
insertToLog(recieveLog,tmp);
RecChar[0]=0;
RecChar[1]='\r';
```

```
                    strcpy(tmp,"");
                    break;
        }
        return 0;
}

void CVICALLBACK MyClearLog (int menuBar, int menuItem, void *callbackData,
                                                    int panel)
{

        strcpy(recieveLog,"");
        strcpy(sendLog,"");
        ResetTextBox (panelMonitorHandle,MONITOR_SendMSG , "");
        InsertTextBoxLine (panelMonitorHandle, MONITOR_SendMSG, 0, sendLog);
        ResetTextBox (panelMonitorHandle,MONITOR_RecieveMSG , "");
        InsertTextBoxLine (panelMonitorHandle, MONITOR_RecieveMSG, 0, recieveLog);
}

void GetCurrentTimeAsString(char *timeStr) {
  time_t currentTime;
  struct tm *localTime;
  time(&currentTime);

  localTime = localtime(&currentTime);

  strftime(timeStr, 50, "%c", localTime);
}

int CVICALLBACK MyRD (int panel, int control, int event,
                                        void *callbackData, int eventData1, int eventData2)
{
```

```
	switch (event)
	{
		case EVENT_COMMIT:
			int pick;
			double tr;
			double delt;
			GetCtrlVal(panel,PANEL_Randow_changes,&pick);
			GetCtrlVal(panel,PANEL_Interval,&tr);
			SectionNode *tmp=sectionHead;

			while(tmp!=NULL)
			{
				SetCtrlAttribute (panel,tmp->section->updateButton_ctrl , ATTR_CTRL_MODE, !pick);
				SetCtrlAttribute (panel,tmp->section->removeButton_ctrl , ATTR_CTRL_MODE, !pick);
				SetCtrlAttribute (panel,tmp->section->data_ctrl , ATTR_CTRL_MODE, !pick);

				tmp=tmp->next;
			}
			SetCtrlAttribute (panel, PANEL_TIMER_RD, ATTR_INTERVAL, tr);
			SetCtrlAttribute (panel,PANEL_Interval , ATTR_CTRL_MODE, !pick);
			SetCtrlAttribute (panel,PANEL_TIMER_RD , ATTR_ENABLED, pick);



	}
	return 0;
}
int CVICALLBACK MyRDTimer (int panel, int control, int event,
```

```
                                        void *callbackData, int eventData1, int
eventData2)
{
        switch (event)
        {
                case EVENT_TIMER_TICK:
                        double tr;
                        double delt;
                        GetCtrlVal (panelHandle, PANEL_Interval, &tr);
                        GetCtrlVal (panelHandle, PANEL_Delta, &delt);
                        int max=number_of_section;
                        SectionNode *tmpHead=sectionHead;
                        SectionNode *tmp=sectionHead;
                        CmtGetLock(sensor_lock);
                        Node * tmpSesHead=sensorHead;
                        CmtReleaseLock(sensor_lock);

                                while(tmp!=NULL)
                                {
                                        CmtGetLock(sensor_lock);
                                        double addVal=getSensorData(tmpSesHead,tmp-
>section->id_sensor);
                                        CmtReleaseLock(sensor_lock);
                                        addVal=(double)rand() / RAND_MAX * (2 * delt) +
(addVal - delt);
                                        SetCtrlVal(panelHandle,tmp->section-
>data_ctrl,addVal);
                                        UpdateSensorFromSectionByInx(panelHandle,tmp-
>section->inx,tmpHead,tmpSesHead);
                                        tmp=tmp->next;

                                }
```

```
                        break;
        }
        return 0;
}

void UpdateSectionData(SectionNode *sectionHead,int inx,double data)
{
        SectionNode *tmp=sectionHead;
        while(tmp!=NULL)
        {
                if(tmp->section->inx==inx)
                {
                        SetCtrlVal(panelHandle,tmp->section->data_ctrl,data);
                        return;
                }
                tmp=tmp->next;
        }
}
```