Mathematica Framework for Serial Communication with mbed

The following is a documentation of the Mathematica framework for serial communication with mbed.

The framework uses instruction packets in the form: [Start][ID][Length][Instruction][Parameter 1]...[Parameter N][Checksum]. The packets include command data that mbed will read to know what to do. The packets sent across the serial connection must follow this form, or else mbed would not respond to the messages. The packet includes the following parts (in order):

1. [Start] : The start byte is one byte and notifies the beginning of the packet. The start byte for our framework is the integer 124 or the hex number 0x7C.
2. [ID] : The ID byte is a unique integer value required for serial communication between Mathematica and multiple boards. Each board is assigned a different unique integer value.
3. [Length] : The length byte is the length of the packet. It is calculated as the number of parameters + 2.
4. [Instruction] : Instruction should be either 0 or 1. If instruction is 1, the command would be a read request. This will be a request for information that will be sent back from mbed. If function is 0, the command would be a write request. This will be a request to change some information in mbed.
5. [Parameters] : Parameters depend on the function. If instruction is 1, or a read request, the parameters will determine what information is read from mbed. For example, using “gyro” as an parameter will return some information from the gyroscope. For a read request, the length of parameters is usually 1. If instruction is 0, or a write request, the parameters will specify what to change and the value to change it to. In the case of controlling a snake bot, angles must be passed into servos. Thus, the parameters for a write request are in the form: [Angle1][Angle2]...[AngleN].
6. [Checksum] : The checksum is used to check if the packet is damaged during communication. It is calculated by summing up all the bytes except the checksum itself. If this sum value is greater than the integer value 255 (8 bytes), subtract it by 256. If the checksum calculated does not equal the checksum sent, the packet is destroyed.

The general procedure of sending a request through serial from Mathematica with our framework will now be discussed.

First, the user has to connect to the device with the ConnectDevice function. The argument to this function is the address of the connection to the device. After the connection is made, the user would be able to send “write” messages with WriteMessage. The arguments of the message is passed as an argument to the function. The function will convert the argument into a message form that is acceptable by the framework. ReadMessage works in the same way, except it returns the information that is read from mbed.

Documentation of functions:

ConnectDevice[*DeviceAddress_:* “”, *BaudRate_:* 9600]
Connects through serial to *DeviceAddress*. Saves the connection to a global variable. Sets the baud rate to *BaudRate*.

DeviceAddress depends on the operating system of the computer the device is connected to:
Windows: “COM#”
MacOS or Linux: “/dev/ttyXX” or “/dev/tty.usbserialXX” or something similar

Default value of *WriteMessageArgs*: “” - does nothing.

Default value of *BaudRate*: 9600 - default baud rate supported by most boards.

Examples:
ConnectDevice["/dev/cu.usbmodem1413"]
ConnectDevice["COM4"]

ReadMessage[*ReadMessageArg_:* 0]
Send a read request from the connection global variable with the argument as *ReadMessageArg*. ReadMessage writes an instruction packet with the form [Start][ID][Length][Instruction][Parameter][Checksum] through serial. *ReadMessageArg* is its only parameter. *ReadMessageArg* is an integer which is parsed in mbed. Depending on the integer, mbed can return different information through serial. ReadMessage returns values received from serial after the read request is sent.

Default value of *ReadMessageArg*: 0 - reads all information from the board.

List of available arguments:
0 - reads all information from the board.
2 - gyroscope information
3 - accelerometer information
4 - magnetometer information
5 - IMU calculated angles (roll, pitch, yaw)
6 - all IMU information

Returns a string.

Examples:
ReadMessage[]
ReadMessage[1]

WriteMessage[*WriteMessageArgs_:* “”]

Send a write request to the connection global variable with the arguments as *WriteMessageArgs*. WriteMessage writes an instruction packet with the form [Start][ID][Length][Instruction][Parameter 1]...[Parameter N][Checksum] through serial. *WriteMessageArgs* are its parameters. The arguments are sent one after the other.

WriteMessageArgs is an array of integers or floats.

Default value of *WriteMessageArgs*: “” - nothing is written.

Examples:
WriteMessage[]
WriteMessage[{40}]
WriteMessage[{45,45,45}]
DisconnectDevice[ ]
Disconnects the device that is currently connect. Resets the connection global variable.

Example:
DisconnectionDevice[]

mbed Class for Serial Communication with Mathematica with our protocol

A class for the serial communication was created to ensure that the user have to write as little code as possible for the communication to work. Following is the documentation of the class.

Serial_Communication Class:

Public Functions:
Serial_Communication(PinName tx, PinName rx, int baud_rate);
bool read_packet();
void packet_destroy();

Public Variables:
int packet_data[256];
int packet_size;
int packet_id
Int packet_instruc;

Private Functions:
bool validate_checksum();

Private Variables:
int packet_header
Int packet_length
Int packet_checksum;
int packet_mode;
bool read_mode;

Public Functions:

Serial_Communication(PinName *tx*, PinName *rx*, int *baud_rate*);
Creates a Serial_Communication object. Creates a serial connection with the pins tx and rx. Sets the baud rate of the connection to baud_rate. Initializes the values for all public and private variables.

Parameters:

- **tx** is the pin for the TX line
- **rx** is the pin for the RX line
- **baud_rate** is baud rate for the serial communication. Must be the same as in Mathematica. Default value is 9600.

Example:

```
Serial_Communication sc(USBTX, USBRX, 115200);
// Creates a Serial_Communication object called sc, creates a serial connection with the pins
USBTX and USBRX, and sets the baud rate of the connection to 115200.
```

bool read_packet(int *timeout*);
Reads a packet sent from Mathematica. Function runs until a packet is read, or until function timeouts. Returns true if the packet is read successfully and returns false if packet is broken (checksum mismatch). Also returns false if function timeouts. Destroys the packet before returning false. Sets the public and private variables to values in the packet.

Parameters:

- **timeout** is time in ms till function timeouts and quits. Default value is 100 ms.

Example:

```
state = sc.read_packet(); // reads a packet and returns whether success or fail to state
                          // using default timeout time 100 ms
if(state) { // if success
        // do something with data
}
```

void packet_destroy();
Destroys the current packet. Does so by setting all the public and private variables back to their initial values. Function must be called before reading another packet.

Example:

```
sc.read_packet();
// code to do something with data
sc.packet_destroy();
```

Private Functions:

bool validate_checksum();
Validates the checksum. Calculated in the same way as in Mathematica (summing up all the bytes except the checksum itself). Returns true if checksum is the same as in the packet and returns false otherwise.

Public Variables:

int **packet_data**[256];
Contains the arguments in the packet. The first index corresponds to the read/write type.

int **packet_size**;
Contains the length of the arguments in the packet. Used to traverse the packet_data array.

int **packet_id**;
Contains the ID in the packet.

int **packet_instruc**;
Contains the instruction in the packet. Determines whether request is read or write.

Serial **pc**;
Contains the serial connection. Can be used to output things through serial.

Private Variables:

int **packet_header**;
Contains the header in the packet. Must be equal to 124 to be recognized as a packet.

Int **packet_length**;
Contains the length of the packet. Calculated as the number of parameters + 2.

Int **packet_checksum**;
Contains the checksum of the packet. Calculated in the same way as in Mathematica (summing up all the bytes except the checksum itself).

int **packet_mode**;
Contains the current mode of the packet, which determines what is read next.
0 - ID
1 - Length
2 - Instruction
3 - Args
4 - Checksum

5 - Read complete

bool **read_mode**;
Stores true if header is detected. False otherwise.

The general procedure for receiving a packet and using the data will now be discussed.

First, the user has to create an object of the Serial_Communication class:
Serial_Communication sc(USBTX, USBRX, 115200);
Then, inside the while loop in the main function, the user has to call the read_packet() function to receive a packet:
state = sc.read_packet();
The read_packet() function will return true when read is successful:
if(state) { // do something with data }
How to read and interpret the data is up to the user. An example is shown in Appendix A.
After doing something to the data, always destroy the packet. This must be done before reading another packet:
sc.packet_destroy();

Appendix A - mbed Serial_Communication Class example usage

```
Serial_Communication sc(USBTX, USBRX, 115200); // created Serial_Communication object
int main() { // main function
   while(1) { // main while loop
     state = sc.read_packet(); // read packet with default timeout time 100 ms
      if(state) { // if read successful
        switch(sc.packet_instruc) {
           case 0: // instruction 0. Write
              switch(sc.packet_data[0]) {
                 case 0: // 0. Default
                    sc.pc.printf("Please input an argument to the Write function.");
                    break;
                 case 1: // 1. Servo angles
                    for (int counter = 0; counter < sc.packet_size; ++counter) {
                       servos_list[counter].position((float)(sc.packet_data[counter + 1] - 90));
                    }
                    break;
              }
              break;
           case 1: // instruction 1. Read
              switch(sc.packet_data[0]) {
                 case 1: // 1. Servo angles
                    sc.pc.printf("Nothing to see here.");
                    break;
                 case 2: // 2. Gyro
                    filt_compute(true);
                    sc.pc.printf("%f %f %f", gx, gy, gz);
                    break;
                 case 3: // 3. Accelerometer
                    filt_compute(true);
                    sc.pc.printf("%d %d %d", a[0], a[1], a[2]);
                    break;
                 case 4: // 4. Magnetometer
                    filt_compute(true);
                    sc.pc.printf("%d %d %d", m[0], m[1], m[2]);
                    break;
              }
              break;
        }
        sc.packet_destroy();
      }
   }
}
```