P20 Whiteboard Chat Report

Marcus Corbin
mgc1g18@soton.ac.uk

May 21, 2020

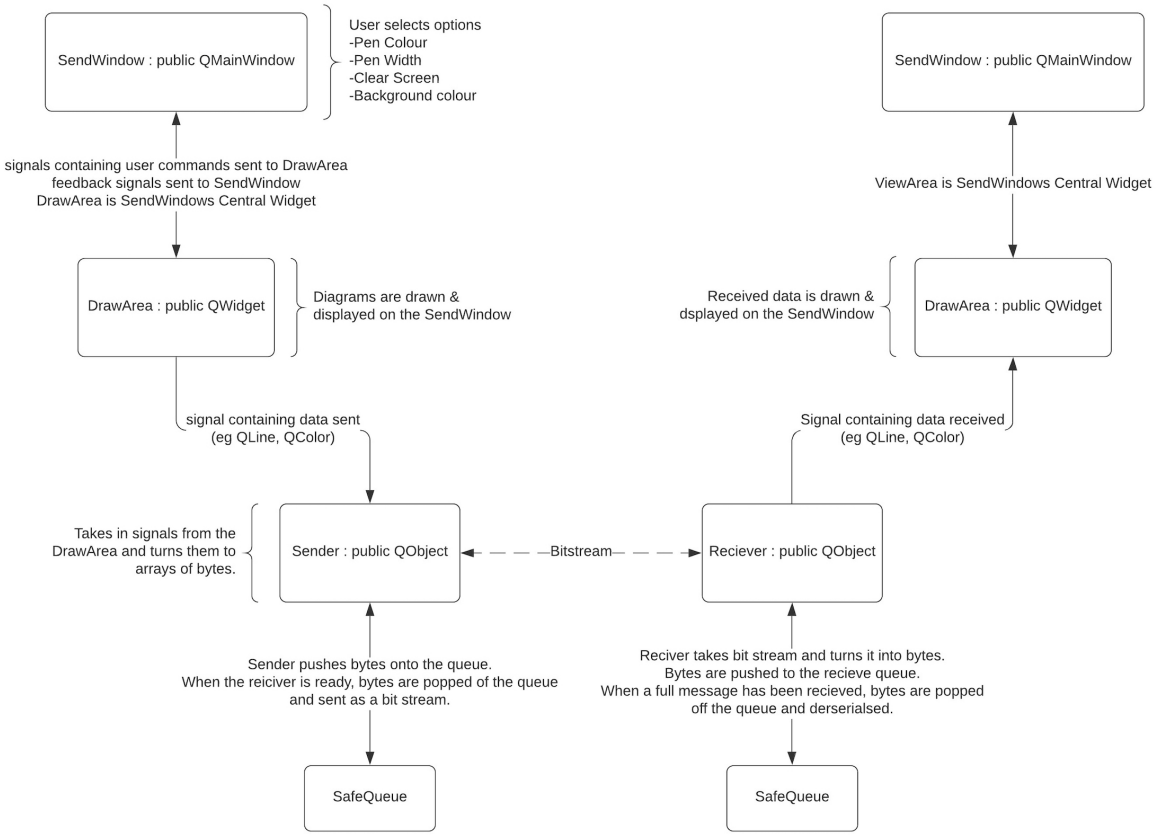

Figure 1: P20 Program Structure

1 Design

1.1 Drawing Diagrams

The `SendWindow` allows users to draw diagrams on the screen by using the `DrawArea` class. This class handles user drawing by overriding the virtual methods `paintEvent`, `mousePressEvent` and `mouseMoveEvent`. The `mousePressEvent` is triggered whenever the user presses the mouse down (within the `DrawArea`). In this method, the current location of the cursor is copied to a variable called `m_OldPoint`. When the user moves the mouse, the `mouseMoveEvent` is triggered and a line is drawn between `m_OldPoint` and the current mouse position, `m_OldPoint` is then updated with the current mouse position, allowing more lines to be drawn with each call of the `mouseMoveEvent` method.

In order to ensure the diagrams are retained when the window is repainted, the lines are not drawn directly to the screen. Instead, the `DrawArea` area class holds an object of type `QPixmap` called `m_Pixmap`. All lines are drawn onto `m_Pixmap`, meaning the window can be repainted at any time and the diagrams will not be lost. Therefore, all the `paintEvent` method has to do is use a `QPainter` to draw the `m_Pixmap` onto the screen.

1.2 Sending Draw Commands Whilst User is Drawing

There are 4 main commands that the program must handle; a draw command, a pen colour command, a pen width command and a clear background command (note the user can also choose a background colour). Every command is handled in the same way. As soon as the user has committed an action, a signal is sent to the `Sender`, the `Sender` then processes that command accordingly depending on the signal that was sent. This way, the user can continuously send commands and all the `DrawArea` has to do is emit the data (e.g for a draw command, `QLine`) and let the `Sender` handle everything else.

1.3 Serializing Commands

As shown in Figure 1, the `Sender` class is responsible for serializing these commands. Figure 2, shows an example of how a 'line' message is serialized.

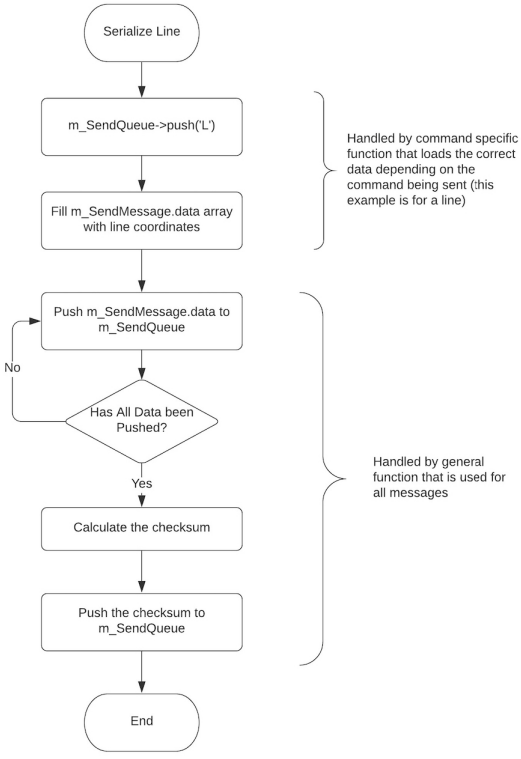

Figure 2: Serialize line flowchart

As noted in the Figure 2, it is only the top two processes that are command specific. Each message has a specific ID, as described in Table 1 and data layout.

Table 1: Table showing ID for each message type

Message Type	ID
Line	'L'
Clear Screen	'C'
Pen Colour	'P'
Pen Width	'W'

After the ID is pushed and the data loaded, actual serialization (shown in the second half of the flowchart), can be done by one general method. This is made possible by the layout of the data type `comms_message_t`.

```
typedef union
{
    uint8_t vals[10];
    struct
    {
    union
    {
            char colour[8];
            int16_t data[4];
    };
    uint16_t checksum;
    };
} comms_message_t;
```

The above data type is the type of `m_SendMessage` in Figure 2. As shown, the data type is 10 bytes long. This means every message we send is 11 bytes, 10 for data and checksum, and 1 for the ID. A message can be made up of either an array of chars that are used to hold a colour or an array of integers that can be used to hold coordinates or a pen width (or any other integer value). As these two arrays take up the same space in memory, one generalized function can be used to push the message to the (m_SendQueue), regardless of the data contained in the `m_SendMessage`. While this may seem overly complicated, it makes the code more readable and makes de-serialization much more simple.

1.4 Using Threads to Send and Receive Messages

In order to send and receive messages while the GUIs' event loop is running, threads must be used. The program therefore creates two extra threads, a send thread and a receive thread. The class `SafeQueue` is a thread safe queue. It uses a mutex to ensure that data is safely passed between threads. Any time data is pushed or popped from the queue, a lock is first acquired and subsequently released. A mutex is also used on the communication booleans to ensure that only one thread can read/write the variables at at time.

1.5 Converting Queued Messages into a Bit Stream

The queued bytes are stored in an instance of the `SafeQueue` class called `m_SendQueue`. The method `void Sender::send()` is responsible for popping bytes off `m_SendQueue` and sending them as a bit stream.

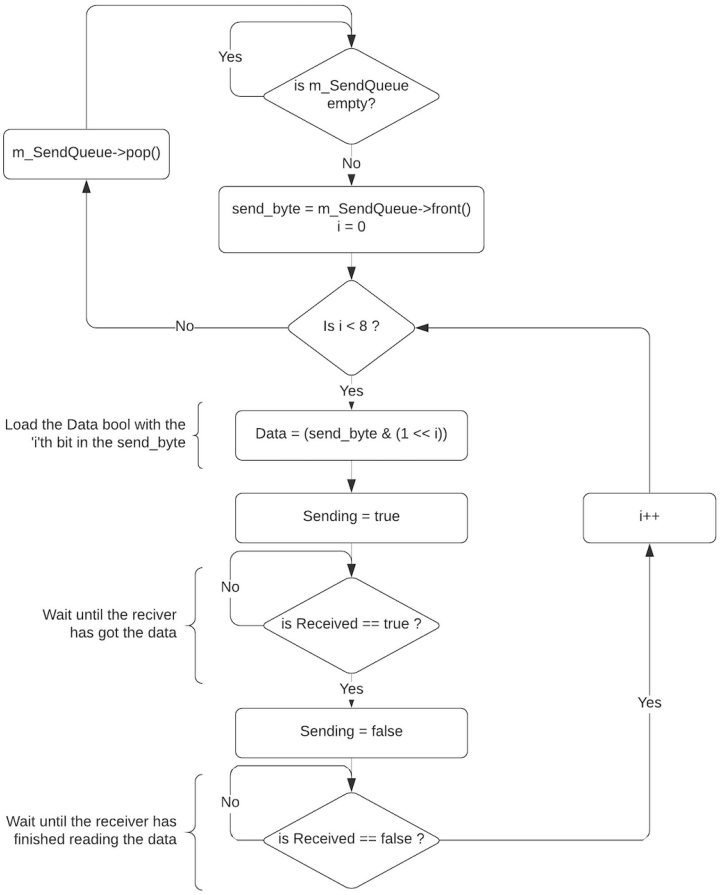

Figure 3: `Sender::send` method flowchart. Note mutex locks and unlocks are not shown

Figure 3 shows how a byte that has been pushed to the send queue is sent as a bit stream. There are three bools used in the communication protocol, in the code they have been put into `namespace Comms`. The bits are: `Comms::sending`, `Comms::received`, `Comms::data`. Two bools were required to interlock the communication in order to stop the `Sender` from continuously sending data. The reason for this is that if real GPIO pins were used, the same pin could not be pulled high and pulled low at the same time as this would result in conflict. This rules out the possibility of using one single flag that is set by the `Sender` and cleared by the `Receiver`.

1.6 Reception and Buffering of the Bit Stream

Just as there is a `Sender` class that handles the serialization and transmission of the bit stream, there is a `Receiver` class that handles the reception and de-serialization of the bit stream. The method `void Receiver::receive()` reads the data bool and constructs a

byte every time 8 bits are received. These bytes are push onto the `m_ReceiveQueue` which is another instance of the `SafeQueue` class. Once the number of received bytes is equal to the number of bytes in a message (11) the bytes are de-serialized.

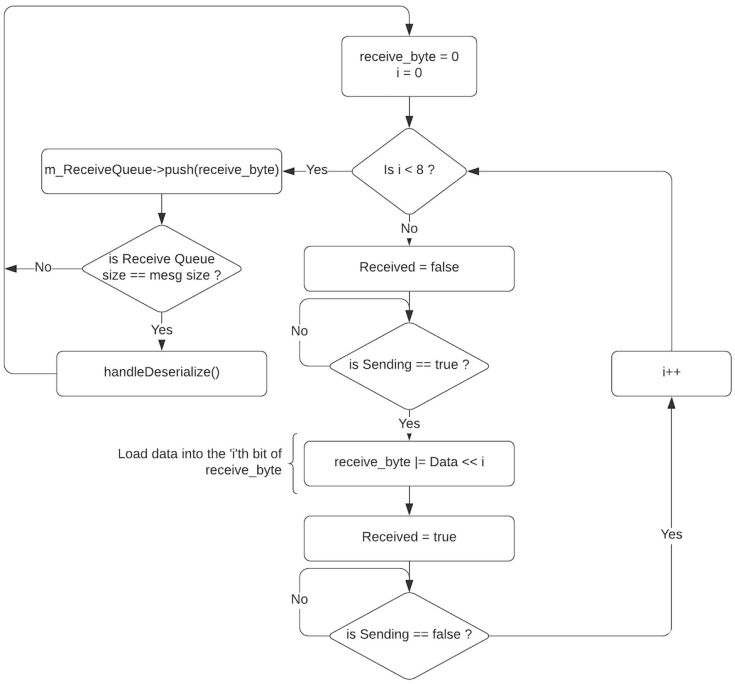

Figure 4: `Receiver::receive` method flowchart. Note mutex locks and unlocks are not shown

Figure 4 shows the process by which bytes are recovered from the bit stream and pushed to the receive queue.

1.7 De-serializing the Received Bytes

As shown in explained in subsection 1.6, once a full message has been received is is de-serialized. This is essentially the opposite process to that in subsection 1.3. The first byte (the ID) of a new message is first checked to see what type it is. Then the message is decoded. This is a general method that fills up the variable `m_ReceiveMessage` (which is of type `comms_message_t`) with data from `m_ReceiveQueue`. The `m_ReceiveMessage .checksum` is then compared with a calculated checksum, generated from the newly received data. If the two checksums match, we have successfully received a message. The next step is to emit a signal to the view area with the reconstructed command (eg a QLine for a line command). If the checksums do not match, `m_ReceiveQueue` is emptied, and that message is ignored.

1.8 Viewing Received Diagrams

Viewing received diagrams is done in a similar way to the way in which users draw commands. Signals that are emitted from the `Receiver` class call different slots depending

on what message was received. The 'draw' message calls a slot that draws a line on the `Viewer` class private member `m_Pixmap`. The colour of the line and its width are chosen by private members `m_PenColour` and `m_PenWidth`. If a 'Pen Colour' message or a 'Pen Width' message is received, slots that change these private members are called. The use of a `QPixmap` means that all the data that is received is stored in the `QPixmap` and the window can therefore be repainted at any time.

2 Implementation

Most of the program has been described in the previous sections however further details of the implementation of various classes is explained below.

2.1 GUI Send & Receive Windows

The two windows are separate classes and inherit the `QMainWindow` class. The `Receive Window` simply sets a `ViewArea` as its 'Central Widget' however the `SendWindow` makes use of some of the `QMainWindow` features. The `SendWindow` has a tool bar which allows the user to select different options such as pen colour, width, clear screen and set background colour. It also has a status bar which shows messages in the bottom left corner of the window. As the user hovers over options on the tool bar it describes their function in the status bar, and when a user starts drawing the "Drawing!" message is displayed.

Figure 5: Screenshot of the toolbar, Icons made by Freepik from www.flaticon.com

Both windows also respond to resize events, meaning the user can resize the window and the content will be scaled accordingly.

2.2 Thread Safe Queue

The class `SafeQueue` is a core part of the program and enables safe transfer of data between threads. The `SafeQueue` class is a template class, allowing a queue of any data type to be setup.

```
template <typename T>
void SafeQueue<T>::push(T data)
{
    pthread_mutex_lock(&m_Mutex);
    m_Queue.push(data);
    pthread_mutex_unlock(&m_Mutex);
}
```

Figure 6: Screenshot of the `SafeQueue::push` method

As shown in Figure 6, the queues mutex `m_Mutex` is locked, the data is pushed, then the mutex is unlocked. This is the case for all the methods in the `SafeQueue` class, and it ensures that only one thread can read/write to/from the queue at a time.

2.3 Communication

The communication system used ensures that no data can be sent before the `Receiver` is ready for it. This means that during test, there was not any errors in the transmission of messages. However as the booleans being used were meant to emulate GPIO pins, a checksum was added to every message to account for real world noise that could interfere with the system.

```
uint16_t checksum16(uint8_t *data, uint16_t len)          // this function generates a 2 byte checksum used at the end of all our messages
{
    uint16_t sum = 0, i;

    for(i=0; i<len; i++)                                  // loop through 'len' bytes
    {
        sum += data[i];                                   // add up all bytes in array, up to length 'len'
    }

    return (~sum);                                        // return the ones compliment
}
```

Figure 7: Screenshot of the `checksum16` function

Figure 7 shows the function that is used to generate the checksums for each message. The function simply loops through the passed in data for length `len` and then returns the result with all the bits flipped (ones compliment).