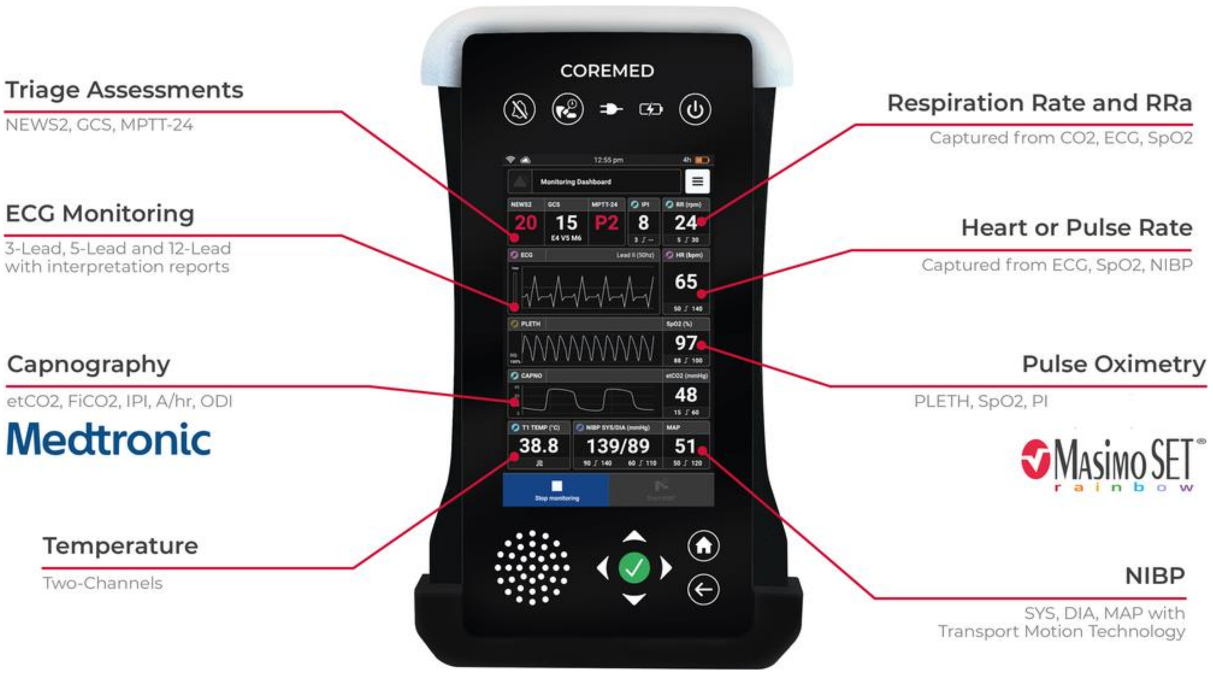
Improving the Service mode application of a medical device



Student ID: B820928
Wolfson School of Engineering
Loughborough University
Words: 5279
21/05/2023

Improving the Service mode application of a medical device

Student ID: B820928
Wolfson School of Engineering
Loughborough University
21/05/2023

Abstract

This dissertation aims to enhance the service mode application of a medical device used for high acuity patient monitoring. The research is bifurcated into two interconnected segments. The initial segment is centred on the presentation of GPS/satellite information to the end-user. In contrast, the subsequent segment is devoted to augmenting the application's scalability by effectively incorporating multi-threading. This two-pronged approach is aimed at maximizing the potential of the device's service mode, rendering it more efficient and responsive for use by maintenance and technician operators alike.

In the first part, the Qt framework's classes are utilized to process and display GPS information, yielding excellent results. However, the satellite information processing required a manual algorithm, as it was not available in the company's Qt version. Although the manual processing produced strong results, there were slight inaccuracies due to the update interval of the satellite information. It was identified that adding unit tests to the manual processing algorithm would be beneficial for future enhancements.

The second part of the research involves transforming the application into a multi-threaded architecture using Qt's QThread object. Two worker threads were created, one dedicated to handling wifi/software updates and the other reused for processing various sections of the GUI. This multi-threading implementation effectively offloaded the workload from the main GUI thread, enhancing responsiveness. Careful consideration and implementation of mutexes were crucial to prevent race conditions. The introduction of multi-threading has made the service mode application more scalable, enabling the seamless integration of additional features without causing GUI lag.

“Overall, this dissertation presents advancements in the service mode application of a medical device, leveraging the Qt framework for GPS processing and implementing multi-threading for improved performance and scalability. The findings contribute to the field of medical device development and highlight potential areas for future research and development efforts” (ChatGPT, 2023) [20].

Table of Contents

Note about figures, tables and/or listings: the section number specifies the section where the figure belongs. For example, "Figure 3.0" means this is the first figure inside section 3 (methodology).

Introduction

Metix Medical is a company in the process of manufacturing a medical device used for high acuity patient monitoring. This device is known as "Coremed", it is a hand-held device aiming to beat its competitors which are larger and not hand-held. The device features a touch screen HMI (Human machine interface). The device runs two applications, the main and the service mode application. The main application would be used for patient monitoring and to be used by the clients (clinicians or paramedics). The service application is to be used by technicians for debugging purposes, it is not to be used by the clients. The main application can be used to monitor patient parameters such as ECG (electrocardiogram), SpO2(oxygen saturation), HR (heart rate), NIBP (non-invasive blood pressure), $ETCO_2$(end-tidal carbon dioxide) just to name a few. Both applications have a GUI (graphical user interface).

"The medical device is powered by a high-performance quad-core embedded processor, which runs an embedded Linux operating system" (ChatGPT, 2023) [20]. The main application is multi-threaded allowing it to utilize all the processor cores to improve performance. A single-threaded application can only utilize a single core. By definition "a thread is a sequence of such instructions within a program that can be executed independently of other code" [1]. The service mode application on the other hand was single threaded.

The service mode application contains views such as SpO2, NIBP and $ETCO_2$. These views display information about the selected parameter and can be used for fault-finding. The application was lacking a section for GPS (global positioning system). The purpose of having a GPS view was to ensure that device location and satellite information can be analysed. The device contains a GPS antenna that can receive signals. When adding a new section for GPS this increases the processing demand of this GUI application. Heavy processing in a single threaded GUI application can lead to an unresponsive application. This warranted the use of a multi-threaded approach to ensure the application is running smoothly. The software of the device is built using the Qt framework. Qt is a popular framework used for developing GUI applications, it supports C++, Python and QML. C++ is a programming language used for building high performance desktop and embedded software, is updated every three years. Most of the device software was written using C++17 and some parts in QML language.

The aim of this dissertation is to discuss how the GPS view was implemented and the multi-threading strategy used to convert a single-threaded application. Specific algorithms, performance and possible short comings will also be discussed. The GPS view required some libraries from the Qt framework, but the satellite information was processed manually without the use of Qt libraries. The manual processing of satellite lacks unit tests which is a limitation, the accuracy of the algorithm was checked manually which was time consuming and prone to human error. Multi-threading was done using Qt's threading support libraries. Debug logs were used to check if the threads were executing as expected as different threads have a different memory address. To thoroughly comprehend the contents of this dissertation, particularly the code examples, it is requisite for the reader to possess a fundamental understanding of object-oriented programming and contemporary C++ practices.

"This dissertation employed the assistance of ChatGPT to refine the grammar and structure of select paragraphs. However, it's crucial to delineate that while ChatGPT played a role in the linguistic and structural enhancement, it did not contribute to the creation of the content itself. All intellectual contributions embodied in this work were exclusively and solely generated by the author" (ChatGPT, 2023) [20].

Methodology

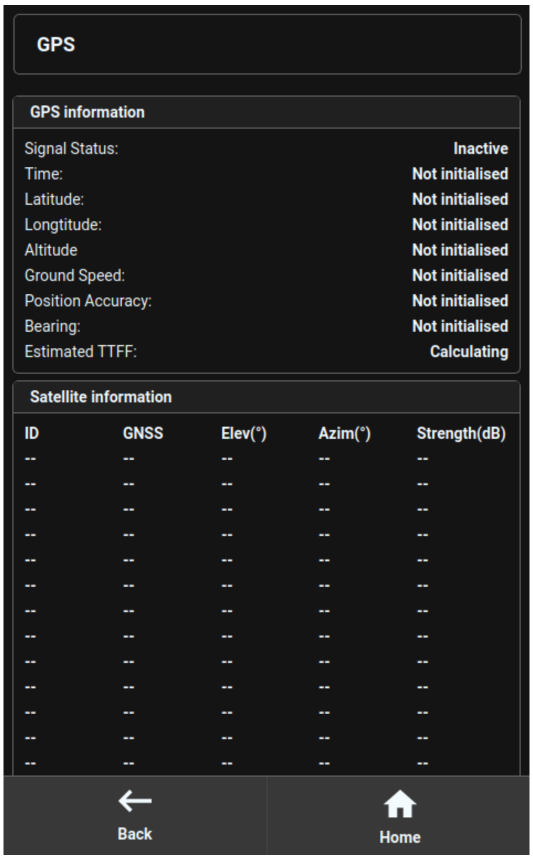
Implementing the GPS view

The acceptance criteria and requirements were decided during team meetings that included senior engineers and the UI/UX (user interface/user experience) team. The GPS view was required to display location information such as time, longitude, latitude, position accuracy, time to first fix (TTFF) and satellite information.

Qt framework offers several classes to process GPS data. One of the classes used to provide GPS was called QNmeaPositionInfoSource[2] which stores GPS in objects of QGeoPositionInfo[5]. See appendix A for an overview of how classes work in the Qt framework.

The device contains a GPS module that detects a GPS signal then sends that data as NMEA strings though a serial port (see appendix C more information about NMEA). The GPS view also required a signal status section which displays whether there is an active or inactive signal. This was implemented using a 1 second timer, if there was not a valid signal after 1 second then the status would be changed to inactive. TTFF was one field that Qt's classes do not provide.

TTFF is an important parameter which describes the time and process required for a GPS device to acquire enough satellite signals to provide accurate navigational information. The word "fix" here means position [3]. Calculating TTFF is quite a complicated process [4] and with the lack of requirement for an accurate TTFF, it was decided to use an estimated TTFF. This "estimated" TTFF would not be an official TTFF but rather the time it took from the device boot until a valid GPS signal. This would be useful for analysing how long it took for the device to get its first valid signal.



GPS information

Signal Status:	Inactive
Time:	Not initialised
Latitude:	Not initialised
Longtitude:	Not initialised
Altitude	Not initialised
Ground Speed:	Not initialised
Position Accuracy:	Not initialised
Bearing:	Not initialised
Estimated TTFF:	Calculating

Satellite information

ID	GNSS	Elev(°)	Azim(°)	Strength(dB)
--	--	--	--	--
--	--	--	--	--
--	--	--	--	--
--	--	--	--	--
--	--	--	--	--
--	--	--	--	--
--	--	--	--	--
--	--	--	--	--
--	--	--	--	--
--	--	--	--	--
--	--	--	--	--
--	--	--	--	--
--	--	--	--	--

Figure 1.0. Wireframe of the GPS view inside the service mode application.

Another parameter that Qt does not directly provide is the position accuracy of the fix, this can be calculated by performing the Pythagoras of horizontal and vertical accuracy as explained in the documentation for QGeoPositionInfo class [5].

Implementing the GPS information in the view was straight forward, data coming from the serial port was processed using the Qt classes and displayed on the view. See appendix C for a detailed code example. On the other hand, implementing the satellite information presented a significant challenge for two reasons. Reason one was the required class QNmeaSatelliteInfoSource was created after Qt version 6.2 whereas the company used an older version 5.12.12. Without access to Qt's class, this meant parsing NMEA data manually. Second reason was that when data is read from the serial port that specific data can no longer be accessed. To implement reason one, serial port data must be copied before it is read by the class that uses it for GPS information. This implementation is shown below. (In an effort to secure the company's confidential intellectual property, most class names have been appropriately prefixed with the term "Dummy"). A class DummySerialPortDemux was created which copies data from the serial port into an array and object of type DummySequentialBuffer.

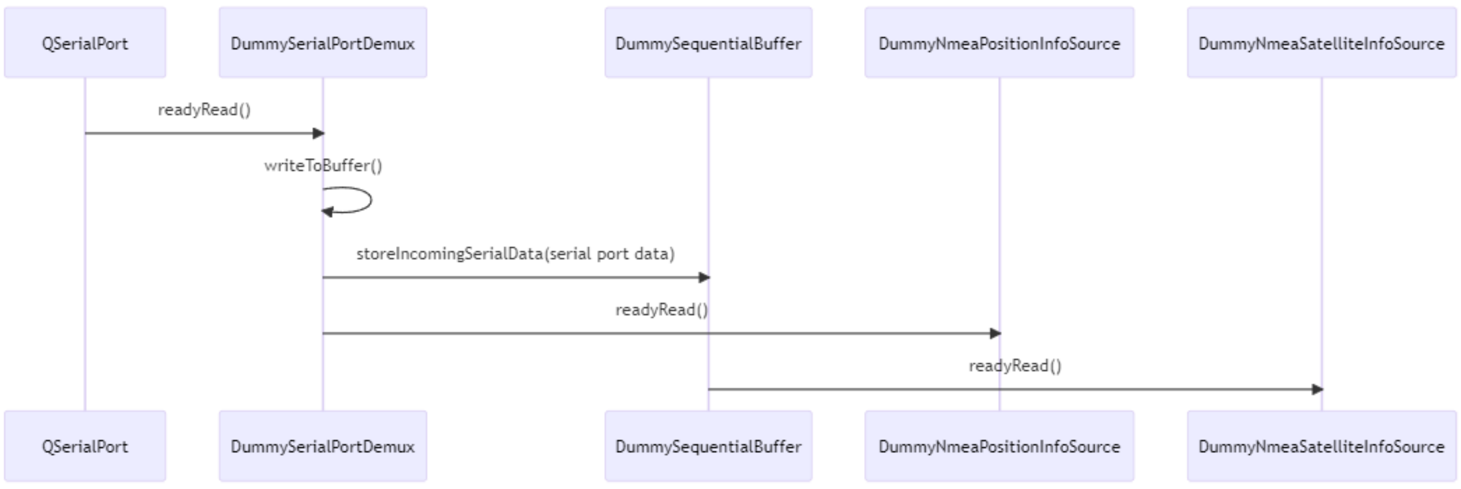


Figure 1.1. UML sequence diagram illustrating how serial port data is copied.

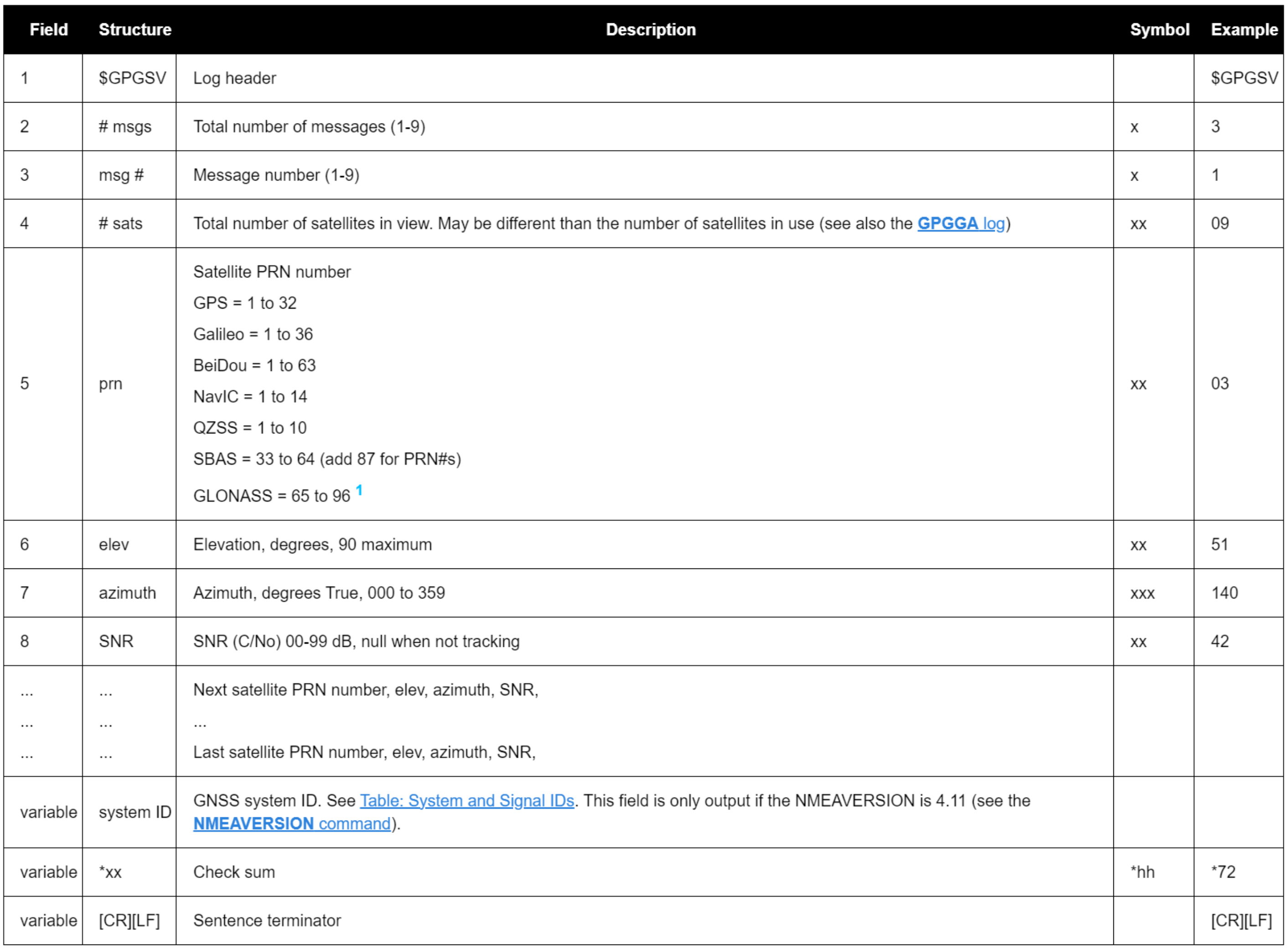
1. readyRead() is a signal emitted by the serial port object. (See appendix A for Qt's signal/slot mechanism).
2. writeToBuffer() saves raw serial port data into an array and inside the DummySequentialBuffer object.
3. Both DummySequentialBuffer and DummySerialPortDemux emit readyRead() signals.

DummySequentialBuffer and DummySerialPortDemux both inherit from the Qt's parent class QIODevice, effectively acting like two serial ports. readyRead() signal is emitted when there is new data available for reading.

DummyNmeaPositionInfoSource classes process NMEA data using Qt's QNmeaPositionInfoSource class hence the similarity in the name. It requires an object of type QIODevice which can emit readyRead() signals therefore DummySerialPortDemux inherits from QIODevice and emits that signal. The reason for using DummyNmeaPositionInfoSource class is TDD (test driven development). The company uses GoogleTest framework for creating unit tests and in this case that requires virtual functions for testing.

When serial port data was successfully copied then it is parsed into satellite information using the class DummyNmeaSatelliteInfoSource. The requirement for this was to display the “satellites in view”, this is useful to check how many satellites are visible to the GPS module. As shown in the wireframe of the GPS view, the company requirements were to display GNSS (Global navigation satellite system), ID (satellite identification number), elevation, azimuth and signal strength (measured in decibels). There are different types of NMEA strings, the necessary string for our scenario was the “GSV” string. The GPS module in the device only supported GPS (US based satellite system) and GLONASS (Russian based satellite system). This means only types of NMEA string were of our interest with regards to satellite information, example strings are shown below. Strings starting with “GP” specify a signal from GPS where “GL” strings specify a GLONASS signal.

```
$GLGSV,3,1,10,81,63,034,51,82,53,272,51,80,52,292,,79,38,200,49*6A

$GPGSV,3,1,09,03,51,140,42,82,53,272,51,80,52,292,,79,38,200,49*72
```



Field	Structure	Description	Symbol	Example
1	$GPGSV	Log header		$GPGSV
2	# msgs	Total number of messages (1-9)	x	3
3	msg #	Message number (1-9)	x	1
4	# sats	Total number of satellites in view. May be different than the number of satellites in use (see also the **GPGGA** log)	xx	09
5	prn	Satellite PRN number GPS = 1 to 32 Galileo = 1 to 36 BeiDou = 1 to 63 NavIC = 1 to 14 QZSS = 1 to 10 SBAS = 33 to 64 (add 87 for PRN#s) GLONASS = 65 to 96 [1]	xx	03
6	elev	Elevation, degrees, 90 maximum	xx	51
7	azimuth	Azimuth, degrees True, 000 to 359	xxx	140
8	SNR	SNR (C/No) 00-99 dB, null when not tracking	xx	42
... ...	... ...	Next satellite PRN number, elev, azimuth, SNR, ... Last satellite PRN number, elev, azimuth, SNR,		
variable	system ID	GNSS system ID. See Table: System and Signal IDs. This field is only output if the NMEAVERSION is 4.11 (see the **NMEAVERSION** command).		
variable	*xx	Check sum	*hh	*72
variable	[CR][LF]	Sentence terminator		[CR][LF]

Figure 1.2. Satellite information inside the NMEA string.

The class DummyNmeaSatelliteInfoSource was developed to parse NMEA strings to objects that contain satellite information. The functions and algorithms were emulated from Qt’s source code for the QNmeaSatelliteInfoSource class. Making use of Qt’s signal/slot mechanism, the data from the serial port was passed into this function parseFromDemux(). The canReadLine() and readLine() perform exactly as their name suggests. The first step in parsing the NMEA is to ensure that the data from the serial port is indeed a valid NMEA string. The function processNmeaData() does this by using a checksum calculator. It initially checks whether the string starts with ‘$’ and contains ‘*’ towards the end of the end of string, the ‘*’ specifies the start of the checksum as shown in Figure 1.2. The NMEA standard specifies that XOR of the bytes between ‘$’ and the ‘*’ is known as the checksum, which is the hexadecimal number provided at the end of the string. The checksum from the C-style string is converted into a QByteArray object and compared with the calculated checksum, this comparison is done by converting the QByteArray into an integer of base 10(note the function argument of 16 is provided specifying the conversion in listing 1.0). The checksum code is emulated from a GitHub repository [6].

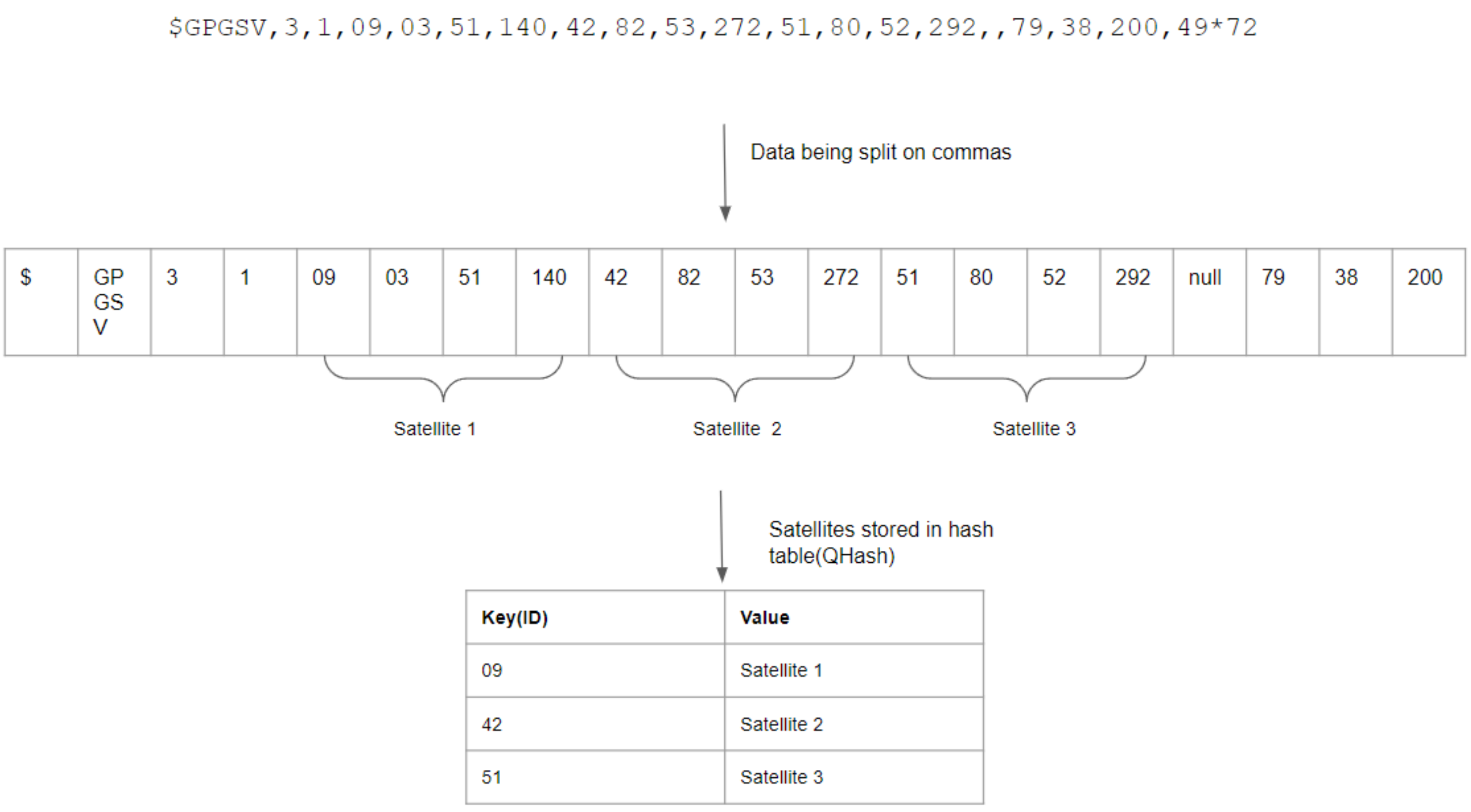
When the checksum is verified, and we have a valid NMEA string we begin to store satellite information in the class QGeoSatelliteInfo. This is done in the following steps:

1. The NMEA string is split into an array (QList) with the comma separators removed.
2. At index 3 of the array, we check if there are more than zero satellites (‘# sats’ in Figure 1.2), if the ‘if’ statement returns true then we move onto step 3. (The toInt() function converts the QByteArray object into an integer type).
3. A for loop with an iterator is used to iterator over the array which contains the data split from the NMEA string.
4. We start at index 4, then use the knowledge from Figure 1.2 to store the appropriate information into the QGeoSatelliteInfo objects. The for loop terminates when we encounter the ‘*’ symbol. Just to make sure there are no infinite loops, we put another condition to end the loop if the iterator reaches the end of string.

Important part of the algorithm in code is shown below:

```
 DummyNmeaSatelliteInfoSource::parsefromDemux(){
if (m_device->canReadLine()){
    while(m_device->canReadLine()){
        char buf[1024];
        int len = m_device->readLine(buf,sizeof(buf));
        processNmeaData(buf,len);
    }
}


 DummyNmeaSatelliteInfoSource::processNmeaData(char* buf, int len){

if (buf[0] == '$' && buf[len-4] == '*'){
    int checksum = nmea0183_checksum(buf);
    QByteArray buf_checksum(&buf[len-3],2);

    if (buf_checksum.toInt(nullptr,16) == checksum){

        if (!strncmp(buf+3,"GSV",3)){
            parseGSV(m_satellites,QByteArray(buf));
        }
    }
}


 DummyNmeaSatelliteInfoSource::parseGSV(QHash<int,QGeoSatelliteInfo>& satellites,
                                        const QByteArray& gsv){

QList<QByteArray> splitter = gsv.split(',');

if (splitter.at(3).toInt()){

    for(auto iter = splitter.constBegin()+=4;
             (iter!=splitter.constEnd()) && (!iter->contains("*"));
             ++iter)
        {
            QGeoSatelliteInfo temp;
            temp.setSatelliteIdentifier(iter->toInt());
            temp.setSatelliteSystem(determineSatelliteSystem(*iter));

            if ((++iter)->contains('*')) break;
            temp.setAttribute(QGeoSatelliteInfo::Elevation,iter->toInt());

            if ((++iter)->contains('*')) break;
            temp.setAttribute(QGeoSatelliteInfo::Azimuth,iter->toInt());

            if ((++iter)->contains('*')) break;
            temp.setSignalStrength(iter->toInt());

            satellites.insert(temp.satelliteIdentifier(), temp);
        }

}
```

Listing 1.0. Code example for the NMEA parsing algorithm.


$GPGSV,3,1,09,03,51,140,42,82,53,272,51,80,52,292,,79,38,200,49*72
Data being split on commas
$
GP GS V
3
1
09
03
51
140
42
82
53
272
51
80
52
292
null
79
38
200
Satellite 1
Satellite 2
Satellite 3
Satellites stored in hash table(QHash)

Key(ID)	Value
09	Satellite 1
42	Satellite 2
51	Satellite 3

Figure 1.3. Visualising the algorithm for parsing NMEA strings.

The data structure used to store the QGeoSatelliteInfo objects was a QHash, this is an associative container that stores data as key-value pairs. This is the equivalent to std::unorderd_map of the C++ STL (standard template library) [7]. This data structure was chosen because it stores the key-value pairs without order (hence the name “unordered_map”), this provides for a very fast insertion of O (1) time complexity [8]. Another problem with storing satellite data was the build-up of older NMEA strings leading to duplicate satellite objects. The satellite ID (also known as PRN number in Figure 1.2) can be used to uniquely identify the satellite objects. QHash stores each key uniquely therefore providing one way to solve this problem.

Another data structure suitable in our case would be QSet. QSet data structure is equivalent to std::unordered_set from the STL [7]. Internally, QSet is implemented using QHash [9]. To use QSet the developer must provide an overloaded equal-to operator (==). Qt’s class QGeoSatelliteInfo already has an overloaded equal-to operator which performs a full member-wise comparison. However, we only want to compare the satellite ID and not the other parameters since they can change over time. For example, a satellite with ID number 20 could have an elevation of 45 degrees but with the newer data from the serial port the elevation might change to 51 degrees. We want to ensure that we do not save the same satellite twice. The pre-defined equal-to operator for the QGeoSatelliteInfo class made the use of QSet unsuitable, therefore QHash was used.

Once all the satellite information has been saved in the QHash object, that data is emitted using a timer. This class DummyNmeaSatelliteInfoSource allows the user to choose how frequently they require satellite information updates. A timer is used to perform this task using Qt’s class QTimer. When the timer times out it then emits the values of the QHash object as an array:

```
    // this connection is made inside the constructor
    connect(&m_updateTimer, &QTimer::timeout, this,
            &DummyNmeaSatelliteInfoSource::emitPendingSatelliteUpdate);

void DummyNmeaSatelliteInfoSource::emitPendingSatelliteUpdate(){
    if (m_satellites.size()){
        emit satellitesInViewUpdated(m_satellites.values());
        m_satellites.clear();
    }
}
```

Listing 1.1. The timeout function for the timer.

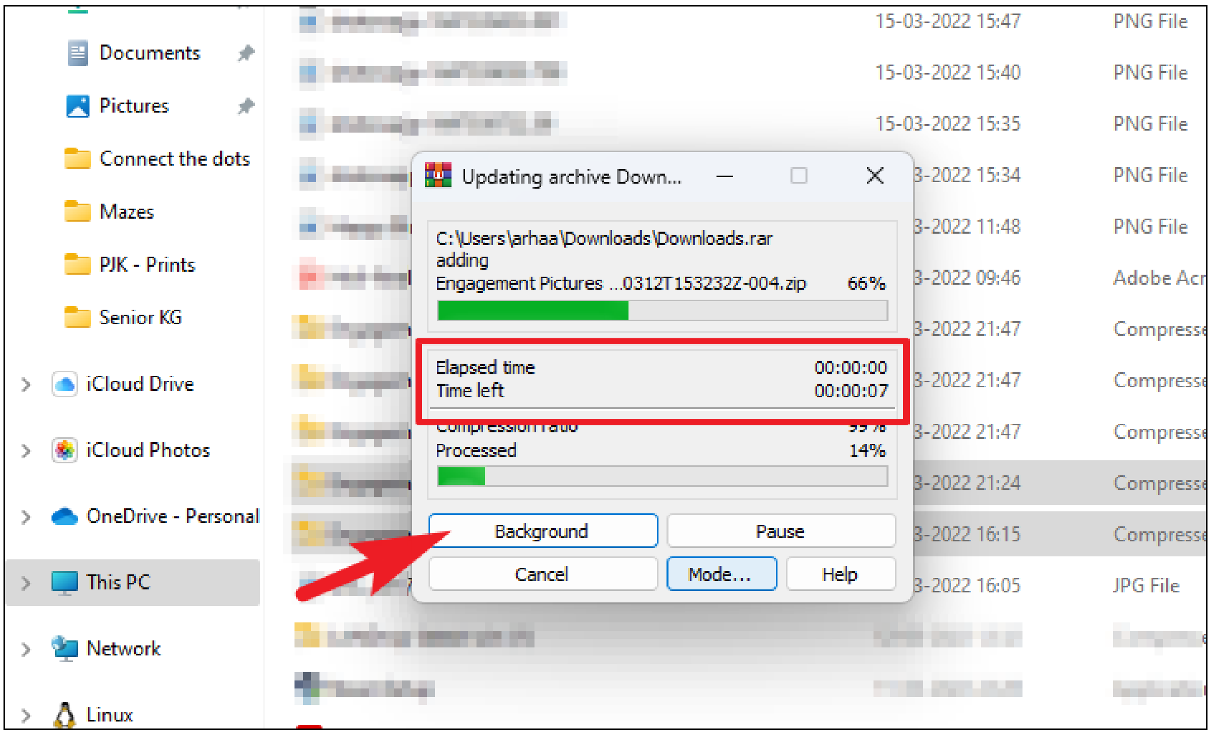
There is not much going on in the function above, it is to illustrate that the satellites are extracted from the QHash(m_satellites) object using the values() function. This timer has an interval set for five seconds. If five seconds pass without any satellite information then the whole satellite information table in Figure 1.0 is set to "--".

Previously we discussed how QNmeaPositionInfoSource was wrapped inside DummyNmeaPositionInfoSource so unit tests could be written. Unit tests are important to ensure the software performs as expected. Due to the time constraints of this project, there were no unit tests developed to test the satellite information parsed from the NMEA data. This is a limitation as any changes to the code base could result in incorrect satellite data being produced. Instead, the satellite information was tested manually using the image shown (Figure 1.2), this was time consuming and inefficient. The major benefit of unit tests is that changes to the code base that inadvertently produce incorrect output can be picked up by the tests. The results were obtained using an NMEA simulator application, this application works by using a timer to send NMEA strings to a serial port. The NMEA strings are read through a text file.

Multi-threading the service mode application

The service mode application was a single threaded application before adding the GPS view. In a single threaded application, the main thread (also known as the GUI thread) is responsible for all the processing. In case the application is performing heavy computations this would make it unresponsive to GUI operations (such as scrolling up and down and/or button clicks) because it is busy performing those computations. This creates a 'laggy' GUI or worse it could freeze the GUI altogether. A simple example of a multi-threaded GUI application would be the windows application "WinRAR". When the user tries to extract compressed files (.zip or .rar and so on), WinRAR displays a progress bar and allows the user to perform other operations on the GUI.

If WinRAR was a single threaded application the entire GUI would freeze until the file extraction is completed, this would obviously frustrate users. This performance improvement justifies the use of multiple threads to prevent freezing the GUI.



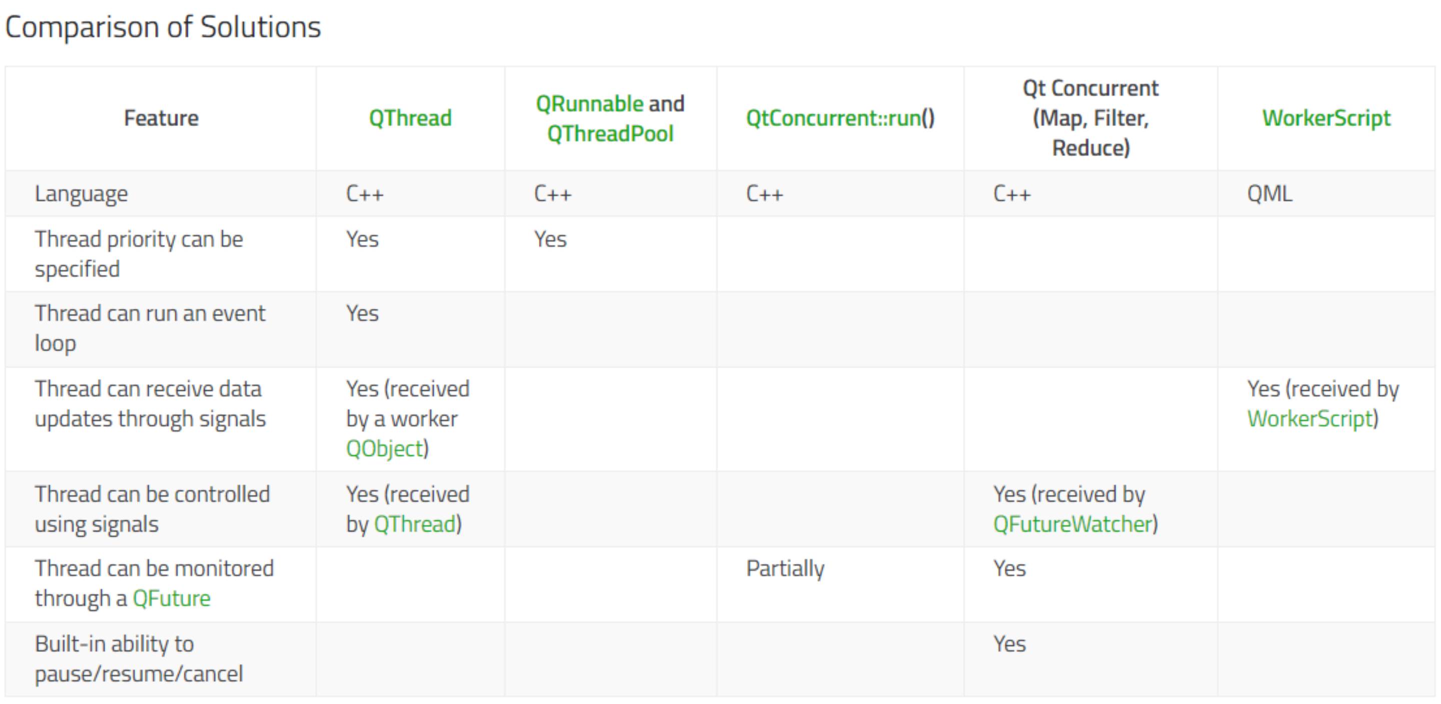
C:\Users\arhaa\Downloads\Downloads.rar
adding
Engagement Pictures ...0312T153232Z-004.zip 66%

Elapsed time 00:00:00
Time left 00:00:07

Processed 14%

Figure 2.0. WinRAR extracting compressed files while the GUI stays responsive [10].

The common multi-threaded strategy deployed in GUI applications is having a separate thread that allows for computations to take place in parallel with the GUI operations. This is explained well on the Qt's documentation where Qt provides a few useful features for multi-threading. A quick comparison can be seen in this table [11].



Comparison of Solutions

Feature	QThread	QRunnable and QThreadPool	QtConcurrent::run()	Qt Concurrent (Map, Filter, Reduce)	WorkerScript
Language	C++	C++	C++	C++	QML
Thread priority can be specified	Yes	Yes			
Thread can run an event loop	Yes				
Thread can receive data updates through signals	Yes (received by a worker QObject)				Yes (received by WorkerScript)
Thread can be controlled using signals	Yes (received by QThread)			Yes (received by QFutureWatcher)	
Thread can be monitored through a QFuture			Partially	Yes	
Built-in ability to pause/resume/cancel				Yes	

Figure 2.1. Table comparing Qt's multi-threaded strategies.

QThread vs std::thread (since C++11)

With the emergence of C++11, a new memory-model was introduced which allowed the user to create threads using std::thread and perform functions using tasks (std::async). Since Qt applications make use of signal/slot mechanism (data can be shared across threads this way) and event loop, std::thread would be unsuitable. Although it (std::thread) can be used to execute functions concurrently, the service mode application requires at least two event loops to run on separate threads which cannot be accomplished using std::thread. Moreover, the use of signal and slots in Qt makes the QThread the most suitable option.

Multi-threading strategy explained 1.

The service mode application on the home page does not require any parallel processing. The processing begins when a certain view is clicked by the user. For example, if the user is on the home page there is no processing taking place but when they click on the GPS view (or any other view) only then will the processing start.

This way we create 1 extra thread at startup, and it is re-used. Creating threads is an expensive operation therefore the consensus is to re-use threads [12].

The initial idea was to have 1 extra thread (known as worker thread) that would be re-used to perform parallel processing with the GUI thread. A GitHub repository called "QtThreadHelper" [13] can be used to create a worker thread that can easily share data with the GUI thread. The repository uses a template header only C++ file which is a wrapper for QThread and also automatically implements a mutex when data is shared across the worker and GUI thread. The advantage of using QtThreadHelper class is how easily it allows data sharing across the GUI thread and worker thread. We examine the "Use case 2" in the readme of the GitHub repository. A quick look inside the guiThread(…) function shows a mutex being used.

```
void MyClass::compute()
{
    // put everything into a worker thread
    workerThread([this]()
        {
            QProgressDialog* progress;
            guiThread(WorkerThread::SYNC, [this, &progress]()
                {
                    progress = new QProgressDialog(this);
                    progress->setWindowModality(Qt::WindowModal);
                });

            for (int i = 0; i < 10; ++i)
            {
                /* do some stuff */
                QThread::sleep(1); // simulate work
                guiThread([progress, i]() { progress->setValue(10 * (i + 1)); });
// needs to be executed in GUI thread
            }

            guiThread([progress]() { delete progress; });
        });
}

// we go deep inside the source code file
template<class Function> inline
void guiThread(WorkerThread::Schedule schedule, Function&& f)
{
    // other code to be ignored...
    switch (schedule)
    {
    case WorkerThread::SYNC:
        {
            QMutex syncMutex;
            syncMutex.lock();   // lock for GUI thread
            QMetaObject::invokeMethod(qGuiApp,
                [f = std::forward<Function>(f), &syncMutex]() mutable
                {
                    std::invoke(std::forward<Function>(f));
                    syncMutex.unlock();
                },
                Qt::QueuedConnection);
            syncMutex.lock();   // wait for GUI call to finish
            syncMutex.unlock();
        }
        break;
    }
}
```

Listing 2.0. Examining the thread_helper.h class's implementation of guiThread(…) function.

The function MyClass::compute() shows how the QtThreadHelper class can be used. Again there is not much going on in the function itself, it uses QThread::sleep(1) to simulate heavy processing then updates a progress bar(similar to WinRAR see Figure 2.0) which is in the GUI thread. The interesting part is in the guiThread(…) function which takes a lambda as a function argument and runs that in the GUI thread while appropriately utilising a mutex. The syntax for the switch/case statement looks verbose but in reality this is just the boiler plate code required to run a certain function when the event loop returns to the object specified.

Implementing this strategy in the company's service mode application would require us to create a worker thread object and re-use that. Using a smart pointer would be suitable here (as using a raw pointer would require us to free the dynamically allocated memory).

We can wrap the functions we want to run in a separate thread in a lambda (similar to the examples shown). This requires some refactoring of the code as all the classes that perform heavy processing require access to the worker thread object. In our case this is passed around by the smart pointer, we use the std::unique_ptr::get() function to get the underlying raw pointer.

If we do not explicitly wrap the required function in the m_workerThread->exec(…) it runs on the GUI thread by default. We can use debug logs (equivalent to printing using std::cout) to check what thread the function is running on. This can be done using the static function QThread::currentThread(). This returns a pointer to the thread where this function resides in, Qt's qCDebug() operator is overloaded to print information about pointers, an example is seen in listing D.0. The debug logs show that this method is useful for checking if the function is indeed running on the desired thread. See appendix D for the C++ implementation (this is interesting but is getting beyond the scope of this dissertation).

Multi-threading strategy explained 2.

The alternative to the above strategy would be to make use of QThread class and add all the necessary objects to the thread. This would allow the same thread to be re-used. To implement this we make use of the QThread::moveToThread(…) function to move the objects to the necessary thread.

To share data from an object with another object on separate thread, we can make use of the Qt::QueuedConnection mechanism. In some classes the need arises where an object method is called from another object which resides in a separate thread. This is where we need to be careful, this is explained in the coming code examples:

```
MainController::MainController(){

    m_workerThread.setObjectName("WorkerThread");
//  GPS/Satellite processing
    m_serialDemux.moveToThread(&m_workerThread);
    m_nmeaGps.moveToThread(&m_workerThread);
    m_nmeaSatellites.moveToThread(&m_workerThread);
//  Module1 processing
    m_module1.moveToThread(&m_workerThread);
//  Module2 processing
    m_module2.moveToThread(&m_workerThread);
//  Module3 processing
    m_module3.moveToThread(&m_workerThread);
//  Software update manager and Wifi processing
    m_wifiThread.setObjectName("WifiThread");
    m_softwareUpdateManager.moveToThread(&m_wifiThread);
    m_wifiModuleManager.moveToThread(&m_wifiThread);
    m_wifiModuleManager.createThreadSafeTimerStopConnection(&m_wifiThread);
//  Start both of the worker threads
    ThreadHelper::startThread(m_workerThread);
    ThreadHelper::startThread(m_wifiThread);
}
```

Listing 3.0. Moving objects to the threads at startup.

The example above shows different modules being added to the worker thread at startup of the application. Initially only one worker thread was used which performed all the non-GUI related operations however, this produced a lag in GPS processing and therefore two new worker threads were created. These two are seen in Listing 3.0 as “WorkerThread” and “WifiThread”.

```
 // Here m_module object (object of type Module1) resides in a separate thread
 void ServiceView1::onShow() {
     m_module1.openPortAndDisplayInformation();
}

 // now we go inside the m_module object
 void Module1::openPortAndDisplayInformation(){
     ThreadHelper::invokeOnThread(this, [this](){

       // perform heavy computations…

     });
 }
```

Listing 3.1. Calling functions from separate threads.

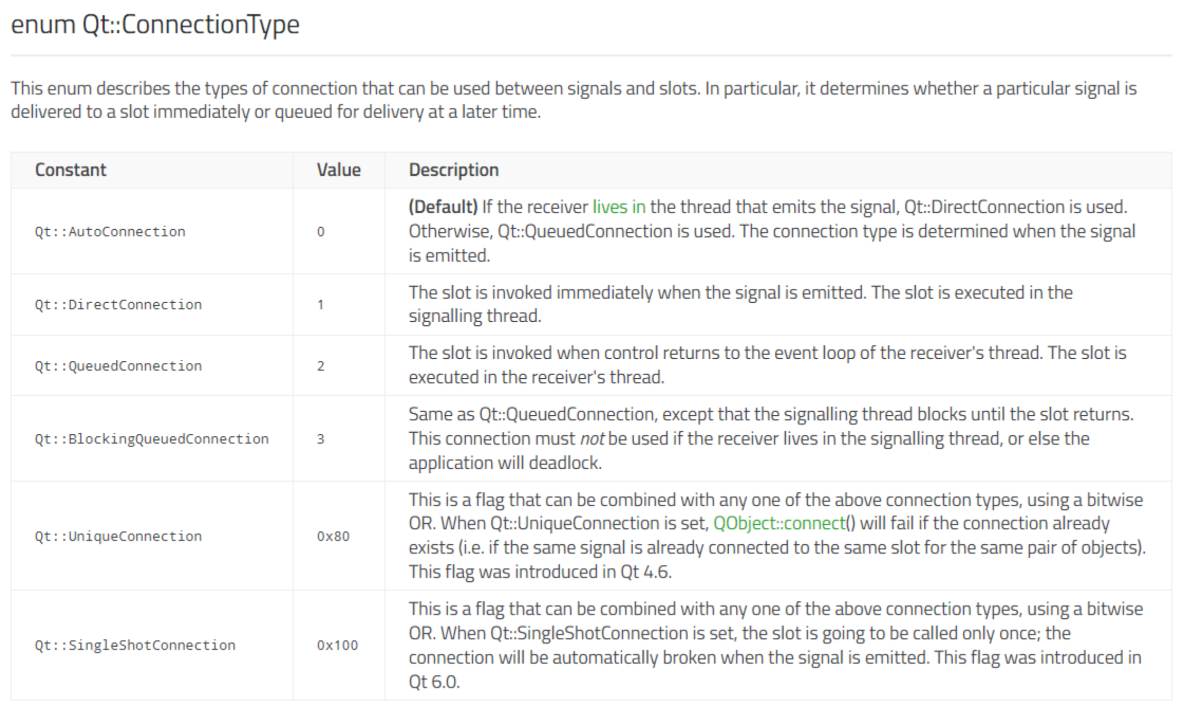
In Listing 3.1 we can see ServiceView1::onShow() calls a function from the m_module1 object which resides on a separate thread(it was moved to the worker thread in Listing 3.0) but the ServiceView1 resides in the GUI thread.

The problem is that the function call would be synchronous, meaning it will be called immediately from the incorrect thread, but we want to ensure it is called in its own thread. To solve this problem, we use ThreadHelper namespace developed by previous engineers at the company. The namespace uses the QMetaObject struct to correctly run the function using Qt::QueuedConnection. This approach is explained on stack overflow [13]. The under the hood implementation of this namespace is explained below:

```
void ThreadHelper::invokeOnThread(QObject *object,
                                  std::function<void ()> &&method) {

    if (object->thread() == QThread::currentThread()) {
        method();
    }
    else {
#if QT_VERSION >= 0x050A00
        QMetaObject::invokeMethod(object, method);
#else
        QTimer::singleShot(0, object, method);
#endif
    }
}
```

Listing 3.2. Looking inside the ThreadHelper namespace.

The first function argument is a pointer to the QObject, the second argument is of type std::function. In Listing 3.1, we pass “this” pointer and a lambda function as the required arguments for this function. The documentation of the QMetaObject struct explaining the usage of the static function invokeMethod(…) states that it invokes the function in the event loop of the QObject passed as the function argument. The invokeMethod(…) function also requires a third argument which is the connection type, in the company code this is not provided and it will default to Qt::AutoConnection(see Figure 3.0 below) and will effectively turn into Qt::QueuedConnection. Qt::AutoConnection decides which connection type to use, this is explained in the documentation.

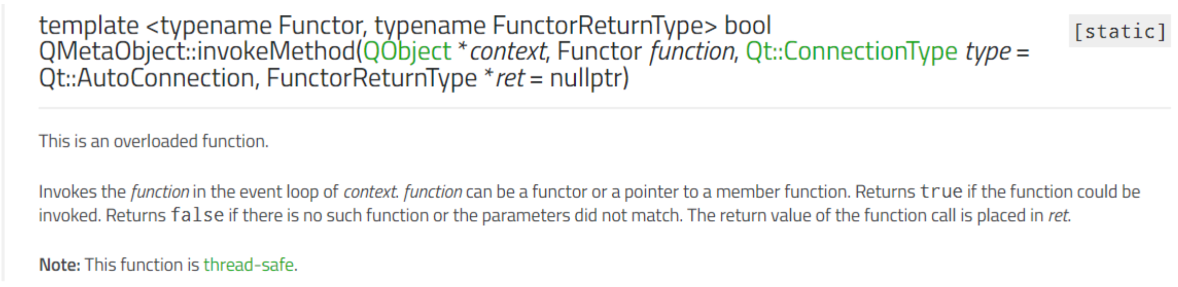


enum Qt::ConnectionType

This enum describes the types of connection that can be used between signals and slots. In particular, it determines whether a particular signal is delivered to a slot immediately or queued for delivery at a later time.

Constant	Value	Description
`Qt::AutoConnection`	0	**(Default)** If the receiver lives in the thread that emits the signal, Qt::DirectConnection is used. Otherwise, Qt::QueuedConnection is used. The connection type is determined when the signal is emitted.
`Qt::DirectConnection`	1	The slot is invoked immediately when the signal is emitted. The slot is executed in the signalling thread.
`Qt::QueuedConnection`	2	The slot is invoked when control returns to the event loop of the receiver's thread. The slot is executed in the receiver's thread.
`Qt::BlockingQueuedConnection`	3	Same as Qt::QueuedConnection, except that the signalling thread blocks until the slot returns. This connection must *not* be used if the receiver lives in the signalling thread, or else the application will deadlock.
`Qt::UniqueConnection`	0x80	This is a flag that can be combined with any one of the above connection types, using a bitwise OR. When Qt::UniqueConnection is set, QObject::connect() will fail if the connection already exists (i.e. if the same signal is already connected to the same slot for the same pair of objects). This flag was introduced in Qt 4.6.
`Qt::SingleShotConnection`	0x100	This is a flag that can be combined with any one of the above connection types, using a bitwise OR. When Qt::SingleShotConnection is set, the slot is going to be called only once; the connection will be automatically broken when the signal is emitted. This flag was introduced in Qt 6.0.

Figure 3.0. Connection type enum explained in Qt's documentation [15].



template <typename Functor, typename FunctorReturnType> bool QMetaObject::invokeMethod(QObject **context*, Functor *function*, Qt::ConnectionType *type* = Qt::AutoConnection, FunctorReturnType **ret* = nullptr) `[static]`

This is an overloaded function.

Invokes the *function* in the event loop of *context*. *function* can be a functor or a pointer to a member function. Returns `true` if the function could be invoked. Returns `false` if there is no such function or the parameters did not match. The return value of the function call is placed in *ret*.

Note: This function is thread-safe.

Figure 3.1. Documentation for the QMetaObject struct's invokeMethod(…) function [16].

The idea is very similar to the signal/slot mechanism, we want to ensure that this function runs when the event loop returns to the specified object's thread. As explained in [14] the function will run asynchronously, meaning it will run when the event loop reaches it.

Another problem that arises with multi-threading application is a race condition. This occurs when more than one thread attempts to access a shared variable. A race condition has undefined behaviour. One way to tackle this problem is to use a mutex. A mutex can be locked and unlocked, we can use a mutex to ensure that only one thread can access the shared variable.

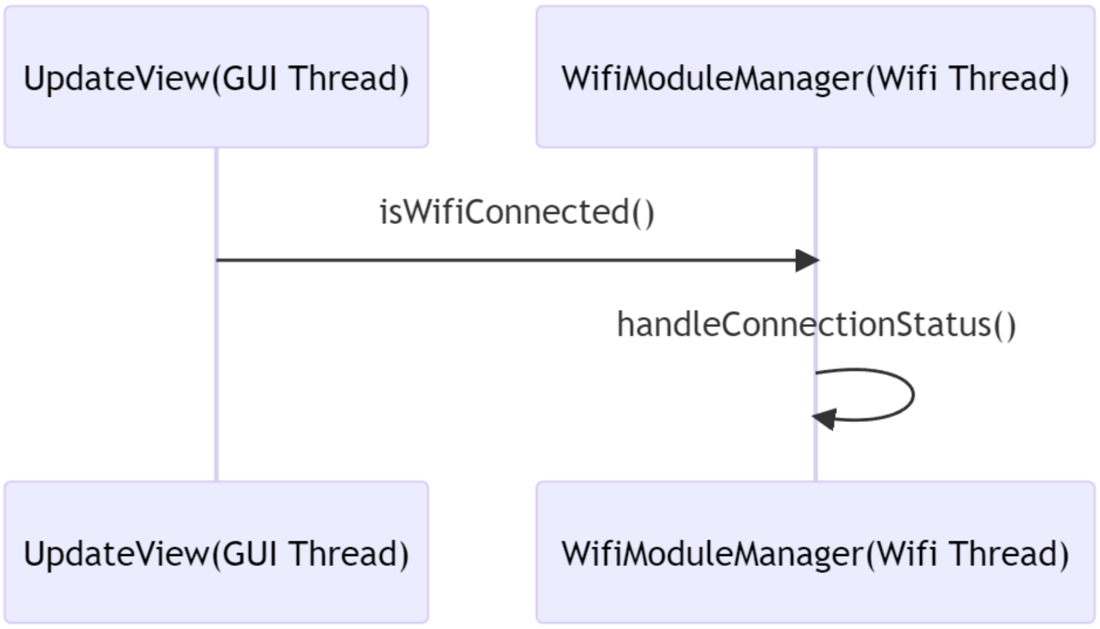


Figure 3.2. UML sequence diagram showing the GUI threading calling a function from WifiManager class.

```
// UpdateView class calls this function from the GUI thread
bool WifiModuleManager::isWifiConnected() const {
    QMutexLocker locker(&m_mutex);
    return m_isWifiConnected;
}

// This function is called within the WifiThread
void WifiModuleManager::handleConnectionStatus() {
    bool connected = false;

// other code…
    m_mutex.lock();
    if (m_isWifiConnected != connected) {
        m_isWifiConnected = connected;
        emit wifiStateChange(m_isWifiConnected);
    }
    m_mutex.unlock();
}
```

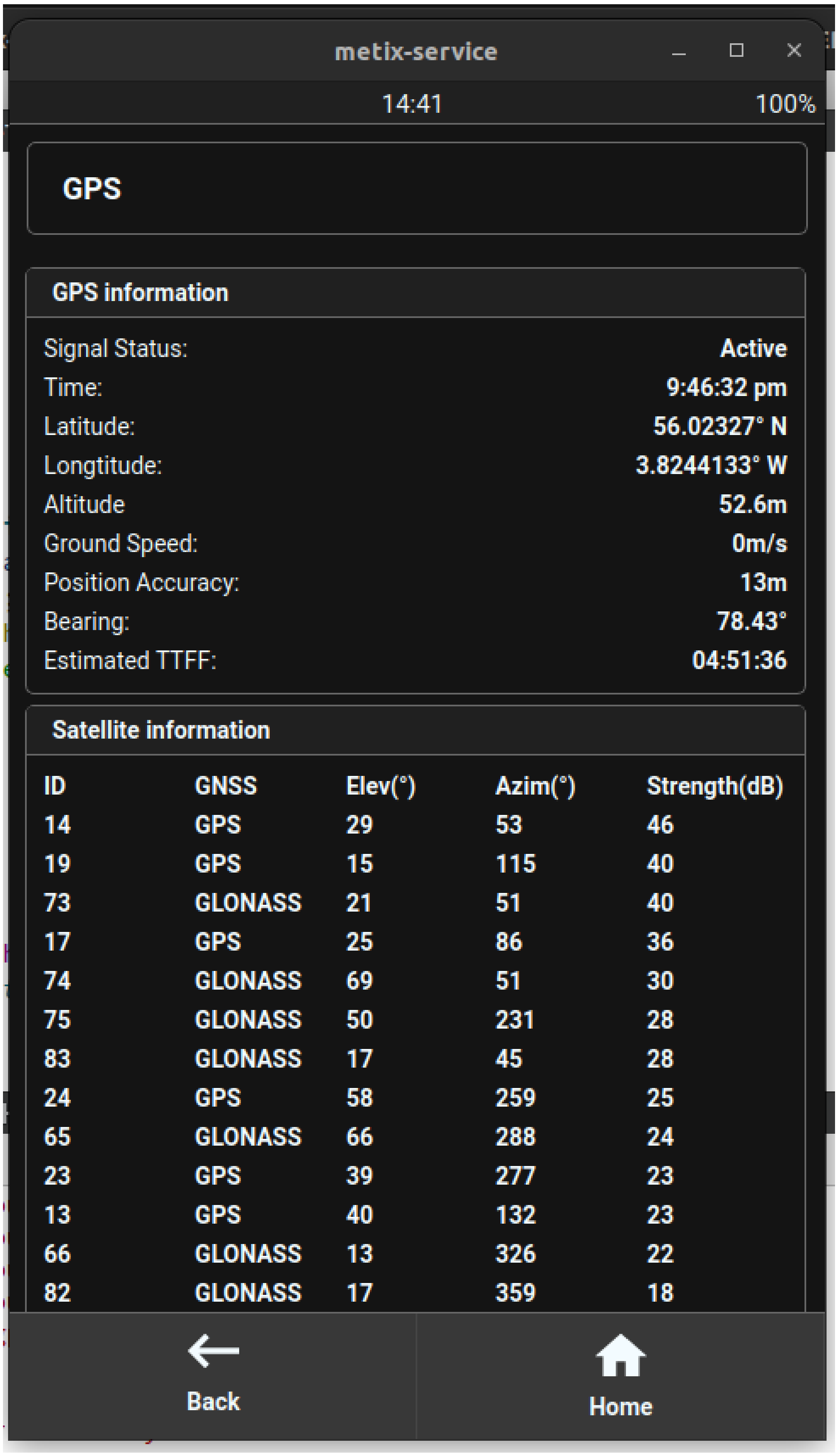
Listing 3.3. Using a mutex to protect the shared variable m_isWifiConnected.

Note the class QMutexLocker will deploy the RAII (resource acquisition is initialisation) technique where the mutex would be locked in the constructor of the object and unlocked in the destructor (when the compiler goes out of scope specified by curly braces). Since the mutex is modified in the const member function this requires the mutex variable to be mutable.

Results

Implementing the GPS View

These results were generated using an NMEA simulator to emulate a real-life GPS signal. The NMEA simulator uses a 1 second timer to write NMEA strings to the specified serial port. The wireframe looks like the following, the signal status is active and the estimated TTFF displays the time it took to get a valid signal.



GPS

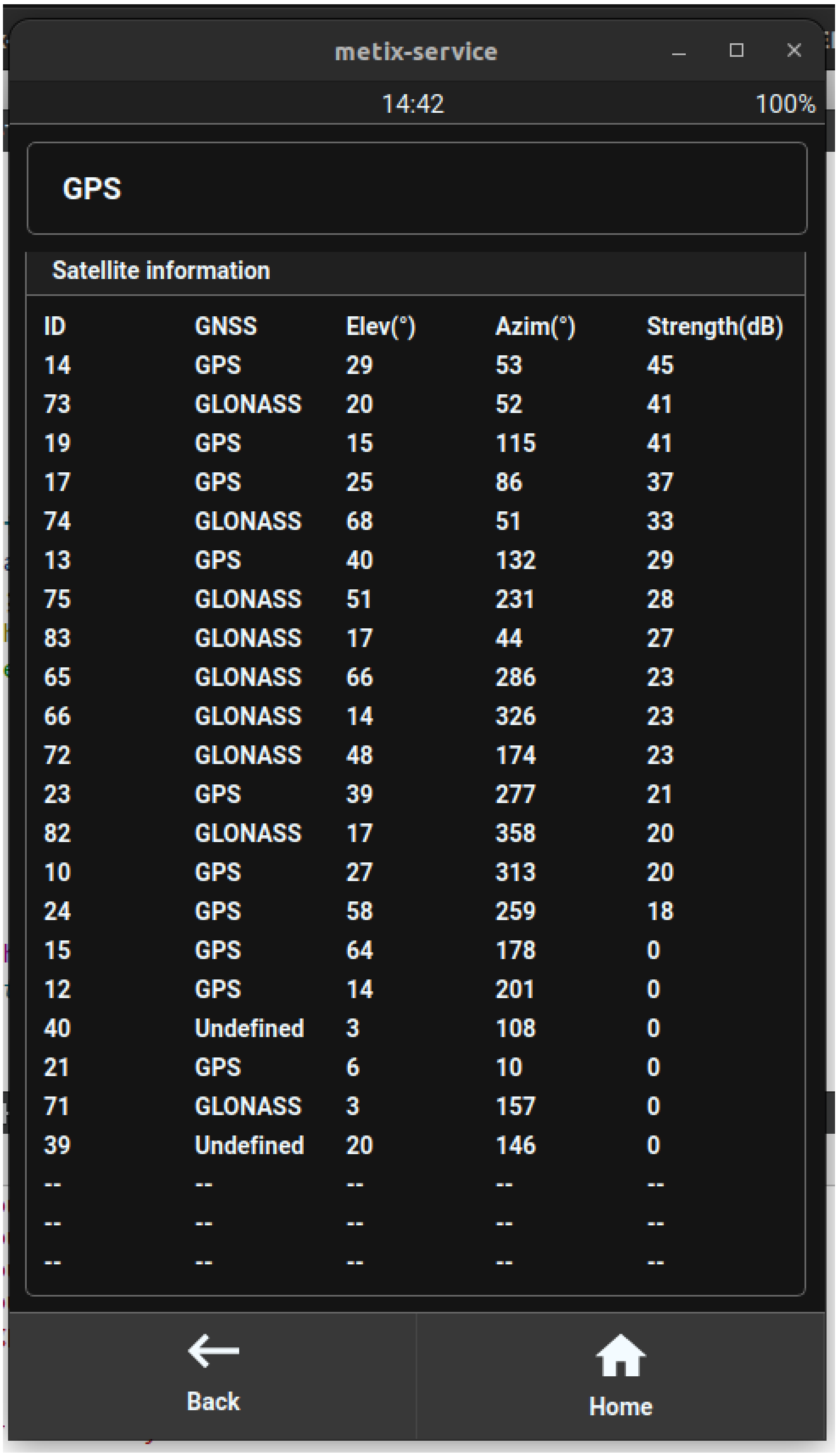
GPS information

Signal Status:	Active
Time:	9:46:32 pm
Latitude:	56.02327° N
Longtitude:	3.8244133° W
Altitude	52.6m
Ground Speed:	0m/s
Position Accuracy:	13m
Bearing:	78.43°
Estimated TTFF:	04:51:36

Satellite information

ID	GNSS	Elev(°)	Azim(°)	Strength(dB)
14	GPS	29	53	46
19	GPS	15	115	40
73	GLONASS	21	51	40
17	GPS	25	86	36
74	GLONASS	69	51	30
75	GLONASS	50	231	28
83	GLONASS	17	45	28
24	GPS	58	259	25
65	GLONASS	66	288	24
23	GPS	39	277	23
13	GPS	40	132	23
66	GLONASS	13	326	22
82	GLONASS	17	359	18

Figure 4.0. The GPS view displaying results by parsing NMEA data.

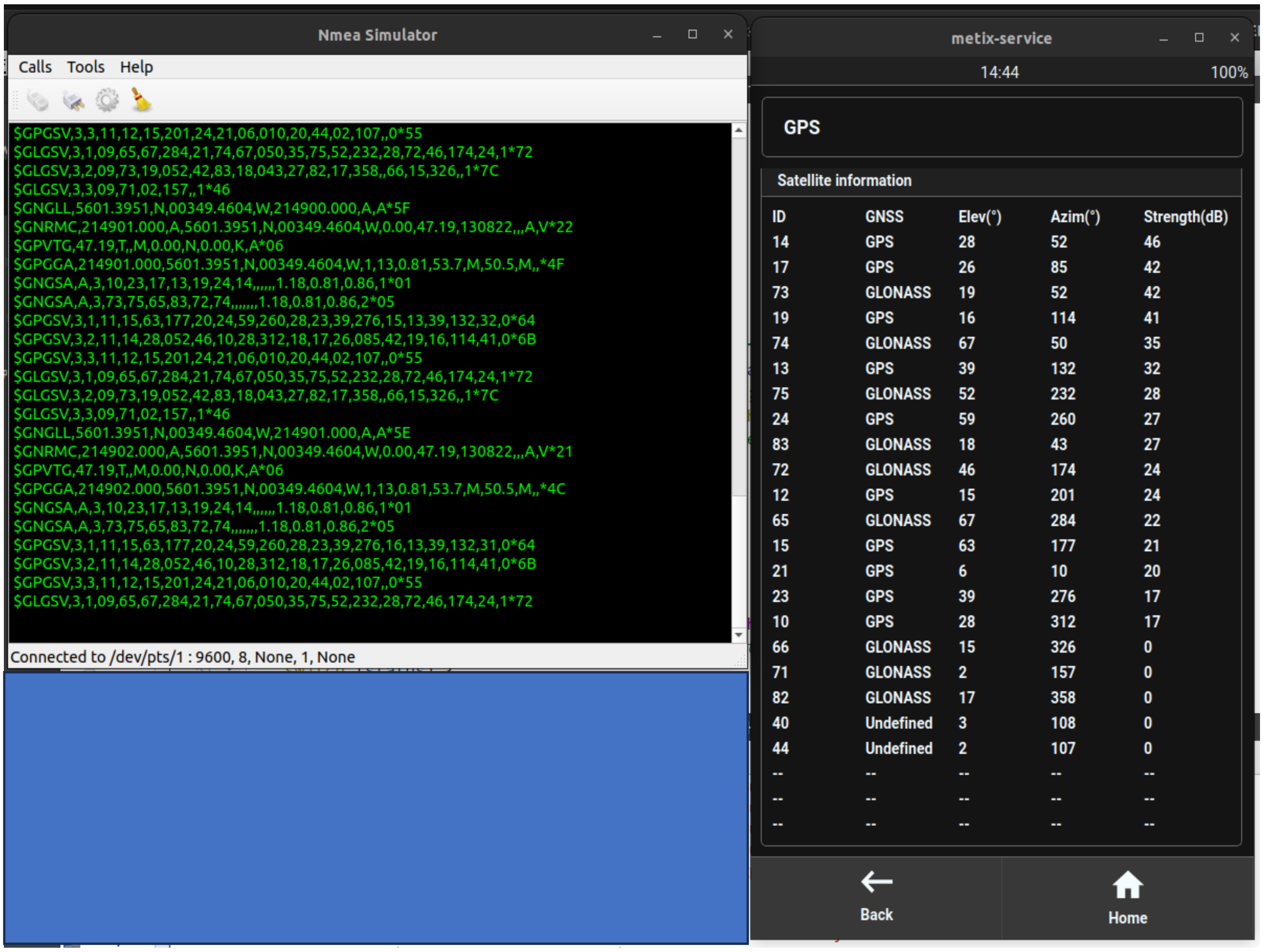


Satellite information

ID	GNSS	Elev(°)	Azim(°)	Strength(dB)
14	GPS	29	53	45
73	GLONASS	20	52	41
19	GPS	15	115	41
17	GPS	25	86	37
74	GLONASS	68	51	33
13	GPS	40	132	29
75	GLONASS	51	231	28
83	GLONASS	17	44	27
65	GLONASS	66	286	23
66	GLONASS	14	326	23
72	GLONASS	48	174	23
23	GPS	39	277	21
82	GLONASS	17	358	20
10	GPS	27	313	20
24	GPS	58	259	18
15	GPS	64	178	0
12	GPS	14	201	0
40	Undefined	3	108	0
21	GPS	6	10	0
71	GLONASS	3	157	0
39	Undefined	20	146	0
--	--	--	--	--
--	--	--	--	--
--	--	--	--	--

Figure 4.1. Satellite information being display from the NMEA data.

We can manually test the satellite information by comparing the NMEA strings with the table provided in Figure 1.2.



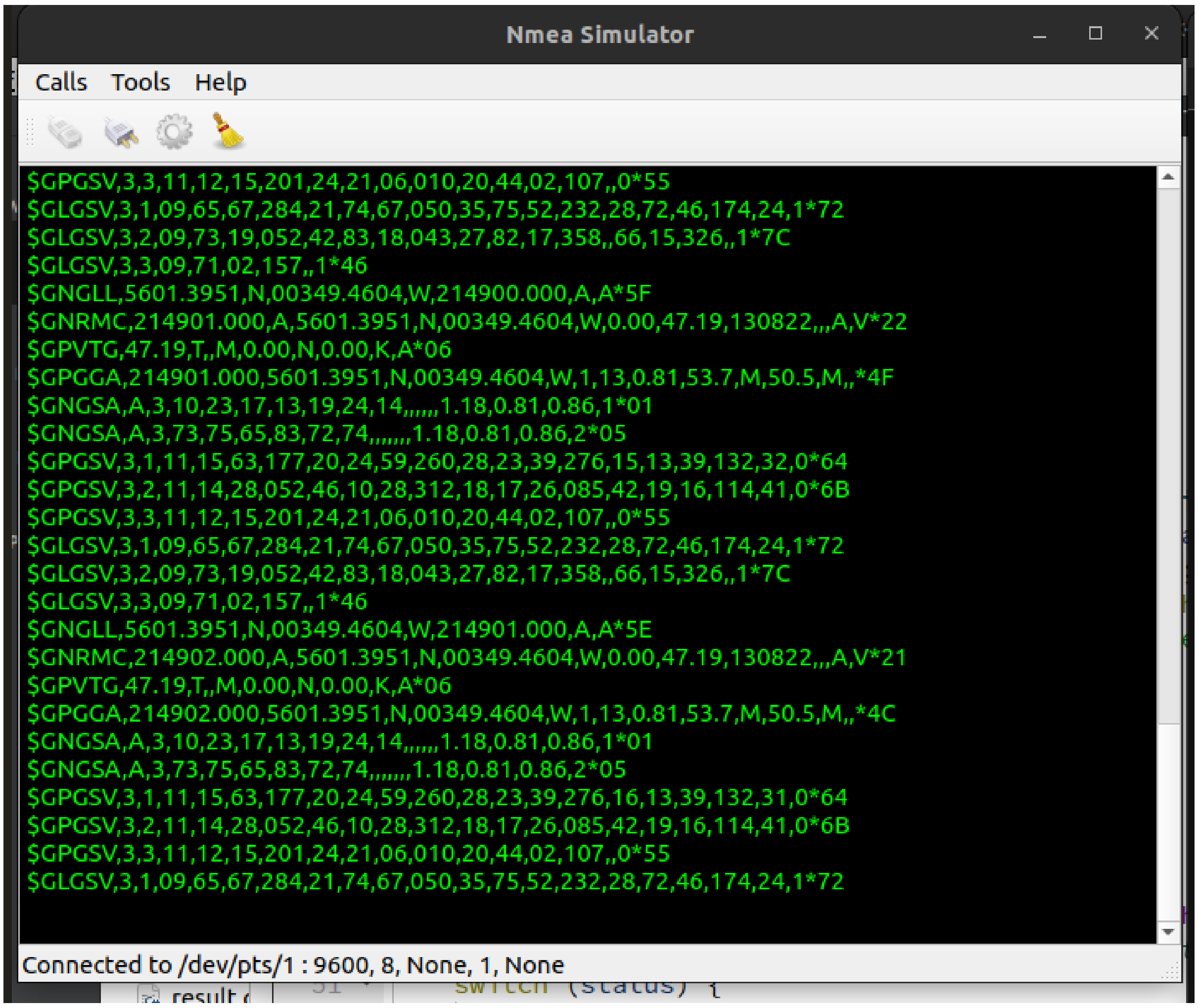
```
$GPGSV,3,3,11,12,15,201,24,21,06,010,20,44,02,107,,0*55
$GLGSV,3,1,09,65,67,284,21,74,67,050,35,75,52,232,28,72,46,174,24,1*72
$GLGSV,3,2,09,73,19,052,42,83,18,043,27,82,17,358,,66,15,326,,1*7C
$GLGSV,3,3,09,71,02,157,,1*46
$GNGLL,5601.3951,N,00349.4604,W,214900.000,A,A*5F
$GNRMC,214901.000,A,5601.3951,N,00349.4604,W,0.00,47.19,130822,,,A,V*22
$GPVTG,47.19,T,,M,0.00,N,0.00,K,A*06
$GPGGA,214901.000,5601.3951,N,00349.4604,W,1,13,0.81,53.7,M,50.5,M,,*4F
$GNGSA,A,3,10,23,17,13,19,24,14,,,,,,1.18,0.81,0.86,1*01
$GNGSA,A,3,73,75,65,83,72,74,,,,,,,1.18,0.81,0.86,2*05
$GPGSV,3,1,11,15,63,177,20,24,59,260,28,23,39,276,15,13,39,132,32,0*64
$GPGSV,3,2,11,14,28,052,46,10,28,312,18,17,26,085,42,19,16,114,41,0*6B
$GPGSV,3,3,11,12,15,201,24,21,06,010,20,44,02,107,,0*55
$GLGSV,3,1,09,65,67,284,21,74,67,050,35,75,52,232,28,72,46,174,24,1*72
$GLGSV,3,2,09,73,19,052,42,83,18,043,27,82,17,358,,66,15,326,,1*7C
$GLGSV,3,3,09,71,02,157,,1*46
$GNGLL,5601.3951,N,00349.4604,W,214901.000,A,A*5E
$GNRMC,214902.000,A,5601.3951,N,00349.4604,W,0.00,47.19,130822,,,A,V*21
$GPVTG,47.19,T,,M,0.00,N,0.00,K,A*06
$GPGGA,214902.000,5601.3951,N,00349.4604,W,1,13,0.81,53.7,M,50.5,M,,*4C
$GNGSA,A,3,10,23,17,13,19,24,14,,,,,,1.18,0.81,0.86,1*01
$GNGSA,A,3,73,75,65,83,72,74,,,,,,,1.18,0.81,0.86,2*05
$GPGSV,3,1,11,15,63,177,20,24,59,260,28,23,39,276,16,13,39,132,31,0*64
$GPGSV,3,2,11,14,28,052,46,10,28,312,18,17,26,085,42,19,16,114,41,0*6B
$GPGSV,3,3,11,12,15,201,24,21,06,010,20,44,02,107,,0*55
$GLGSV,3,1,09,65,67,284,21,74,67,050,35,75,52,232,28,72,46,174,24,1*72
```

Connected to /dev/pts/1 : 9600, 8, None, 1, None

Satellite information

ID	GNSS	Elev(°)	Azim(°)	Strength(dB)
14	GPS	28	52	46
17	GPS	26	85	42
73	GLONASS	19	52	42
19	GPS	16	114	41
74	GLONASS	67	50	35
13	GPS	39	132	32
75	GLONASS	52	232	28
24	GPS	59	260	27
83	GLONASS	18	43	27
72	GLONASS	46	174	24
12	GPS	15	201	24
65	GLONASS	67	284	22
15	GPS	63	177	21
21	GPS	6	10	20
23	GPS	39	276	17
10	GPS	28	312	17
66	GLONASS	15	326	0
71	GLONASS	2	157	0
82	GLONASS	17	358	0
40	Undefined	3	108	0
44	Undefined	2	107	0
--	--	--	--	--
--	--	--	--	--
--	--	--	--	--

Figure 4.2. The NMEA strings alongside the satellite data that was extracted.



```
$GPGSV,3,3,11,12,15,201,24,21,06,010,20,44,02,107,,0*55
$GLGSV,3,1,09,65,67,284,21,74,67,050,35,75,52,232,28,72,46,174,24,1*72
$GLGSV,3,2,09,73,19,052,42,83,18,043,27,82,17,358,,66,15,326,,1*7C
$GLGSV,3,3,09,71,02,157,,1*46
$GNGLL,5601.3951,N,00349.4604,W,214900.000,A,A*5F
$GNRMC,214901.000,A,5601.3951,N,00349.4604,W,0.00,47.19,130822,,,A,V*22
$GPVTG,47.19,T,,M,0.00,N,0.00,K,A*06
$GPGGA,214901.000,5601.3951,N,00349.4604,W,1,13,0.81,53.7,M,50.5,M,,*4F
$GNGSA,A,3,10,23,17,13,19,24,14,,,,,,1.18,0.81,0.86,1*01
$GNGSA,A,3,73,75,65,83,72,74,,,,,,,1.18,0.81,0.86,2*05
$GPGSV,3,1,11,15,63,177,20,24,59,260,28,23,39,276,15,13,39,132,32,0*64
$GPGSV,3,2,11,14,28,052,46,10,28,312,18,17,26,085,42,19,16,114,41,0*6B
$GPGSV,3,3,11,12,15,201,24,21,06,010,20,44,02,107,,0*55
$GLGSV,3,1,09,65,67,284,21,74,67,050,35,75,52,232,28,72,46,174,24,1*72
$GLGSV,3,2,09,73,19,052,42,83,18,043,27,82,17,358,,66,15,326,,1*7C
$GLGSV,3,3,09,71,02,157,,1*46
$GNGLL,5601.3951,N,00349.4604,W,214901.000,A,A*5E
$GNRMC,214902.000,A,5601.3951,N,00349.4604,W,0.00,47.19,130822,,,A,V*21
$GPVTG,47.19,T,,M,0.00,N,0.00,K,A*06
$GPGGA,214902.000,5601.3951,N,00349.4604,W,1,13,0.81,53.7,M,50.5,M,,*4C
$GNGSA,A,3,10,23,17,13,19,24,14,,,,,,1.18,0.81,0.86,1*01
$GNGSA,A,3,73,75,65,83,72,74,,,,,,,1.18,0.81,0.86,2*05
$GPGSV,3,1,11,15,63,177,20,24,59,260,28,23,39,276,16,13,39,132,31,0*64
$GPGSV,3,2,11,14,28,052,46,10,28,312,18,17,26,085,42,19,16,114,41,0*6B
$GPGSV,3,3,11,12,15,201,24,21,06,010,20,44,02,107,,0*55
$GLGSV,3,1,09,65,67,284,21,74,67,050,35,75,52,232,28,72,46,174,24,1*72
```

Connected to /dev/pts/1 : 9600, 8, None, 1, None

Figure 4.3. Zoomed in screenshot of the NMEA simulator.

Here we pay attention to the last few GSV strings from Figure 4.3:

$GPGSV,3,1,11,15,63,177,20,24,59,260,28,23,39,276,16,13,39,132,31,0*64

$GPGSV,3,2,11,14,28,052,46,10,28,312,18,17,26,085,42,19,16,114,41,0*6B

$GPGSV,3,3,11,12,15,201,24,21,06,010,20,44,02,107,,0*55

$GLGSV,3,1,09,65,67,284,21,74,67,050,35,75,52,232,28,72,46,174,24,1*72

$GLGSV,3,2,09,73,19,052,42,83,18,043,27,82,17,358,,66,15,326,,1*7C

$GLGSV,3,3,09,71,02,157,,1*46

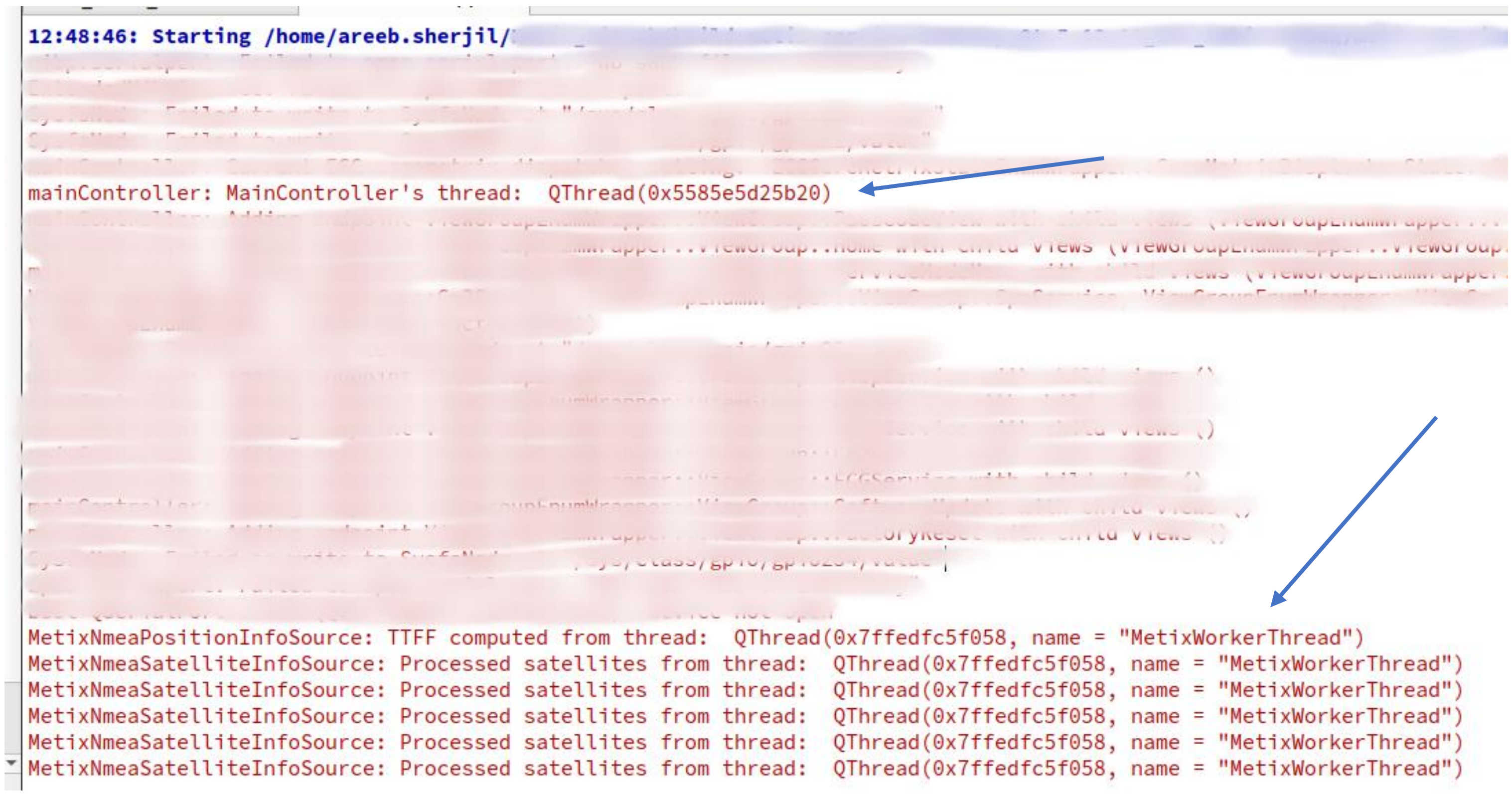
ID	GNSS	Elevation(degrees)	Azimuth(degrees)	Strength(dB)
14		28	52	46
17		26	85	42
73		19	52	42
19		16	114	41
74		67	50	35
13		39	132	31
24		59	260	28
75		52	232	28
83		18	43	27
12		15	201	24
72		46	174	24
65		67	284	21
15		63	177	20
21		6	10	20
10		28	312	18
23		39	276	16
44		2	107	0
82		17	358	0
66		15	326	0
71		2	157	0

Table 4.0. Manually parsing the GSV strings.

The green entries show the parameters that match with Figure 4.2 and red entries are the ones that do not match. The entries are sorted by signal strength in descending order, similar to how they are sorted in Figure 4.2. We ignore the GNSS section because it is not part of the parsing process. These results show that some of the values are unexplained, this will be discussed in more detail in the Discussions section.

Multi-threading the service mode application

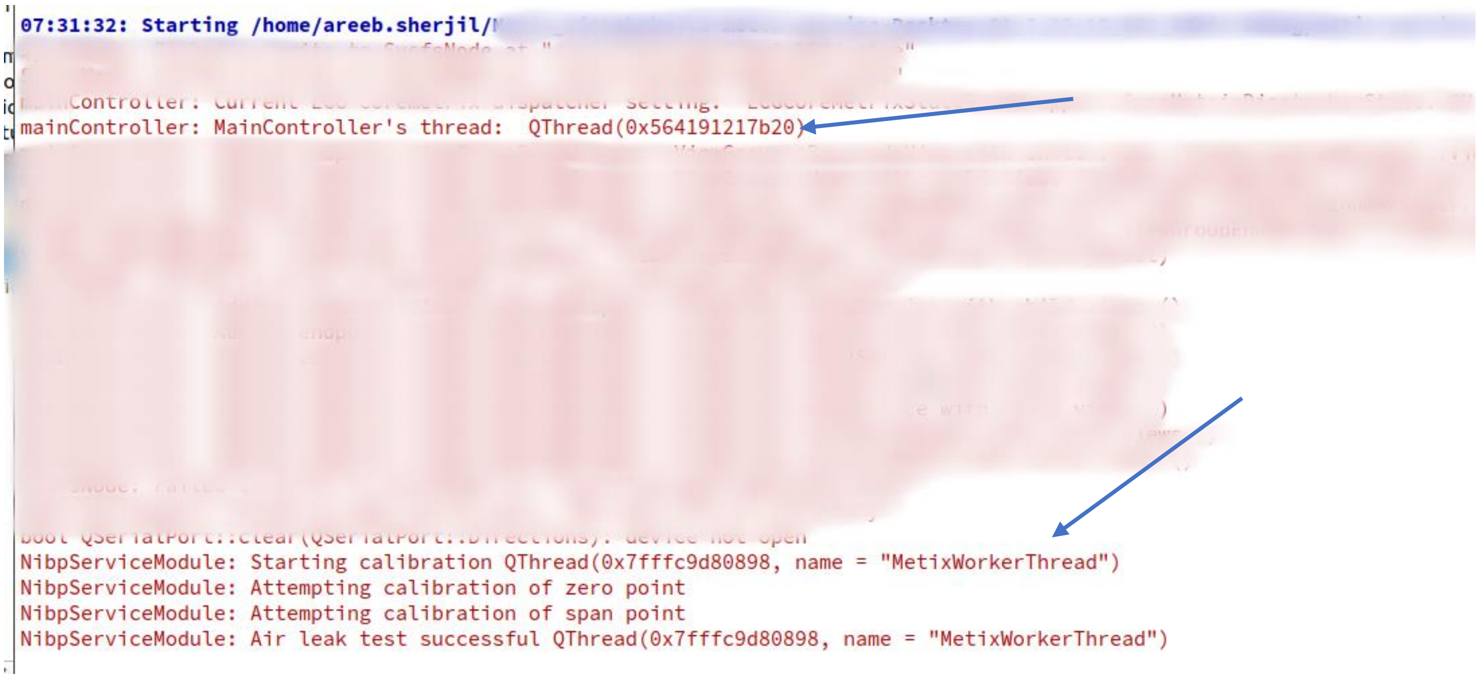
To check whether functions are running on the correct thread we make use of the static function from QThread class which is currentThread(). This function returns a pointer to the thread where it was called from. In Qt the debug operator is overloaded so that it can print out information about pointers.



```
12:48:46: Starting /home/areeb.sherjil/[redacted]
[redacted]
mainController: MainController's thread:  QThread(0x5585e5d25b20)
[redacted]
MetixNmeaPositionInfoSource: TTFF computed from thread:  QThread(0x7ffedfc5f058, name = "MetixWorkerThread")
MetixNmeaSatelliteInfoSource: Processed satellites from thread:  QThread(0x7ffedfc5f058, name = "MetixWorkerThread")
MetixNmeaSatelliteInfoSource: Processed satellites from thread:  QThread(0x7ffedfc5f058, name = "MetixWorkerThread")
MetixNmeaSatelliteInfoSource: Processed satellites from thread:  QThread(0x7ffedfc5f058, name = "MetixWorkerThread")
MetixNmeaSatelliteInfoSource: Processed satellites from thread:  QThread(0x7ffedfc5f058, name = "MetixWorkerThread")
MetixNmeaSatelliteInfoSource: Processed satellites from thread:  QThread(0x7ffedfc5f058, name = "MetixWorkerThread")
```

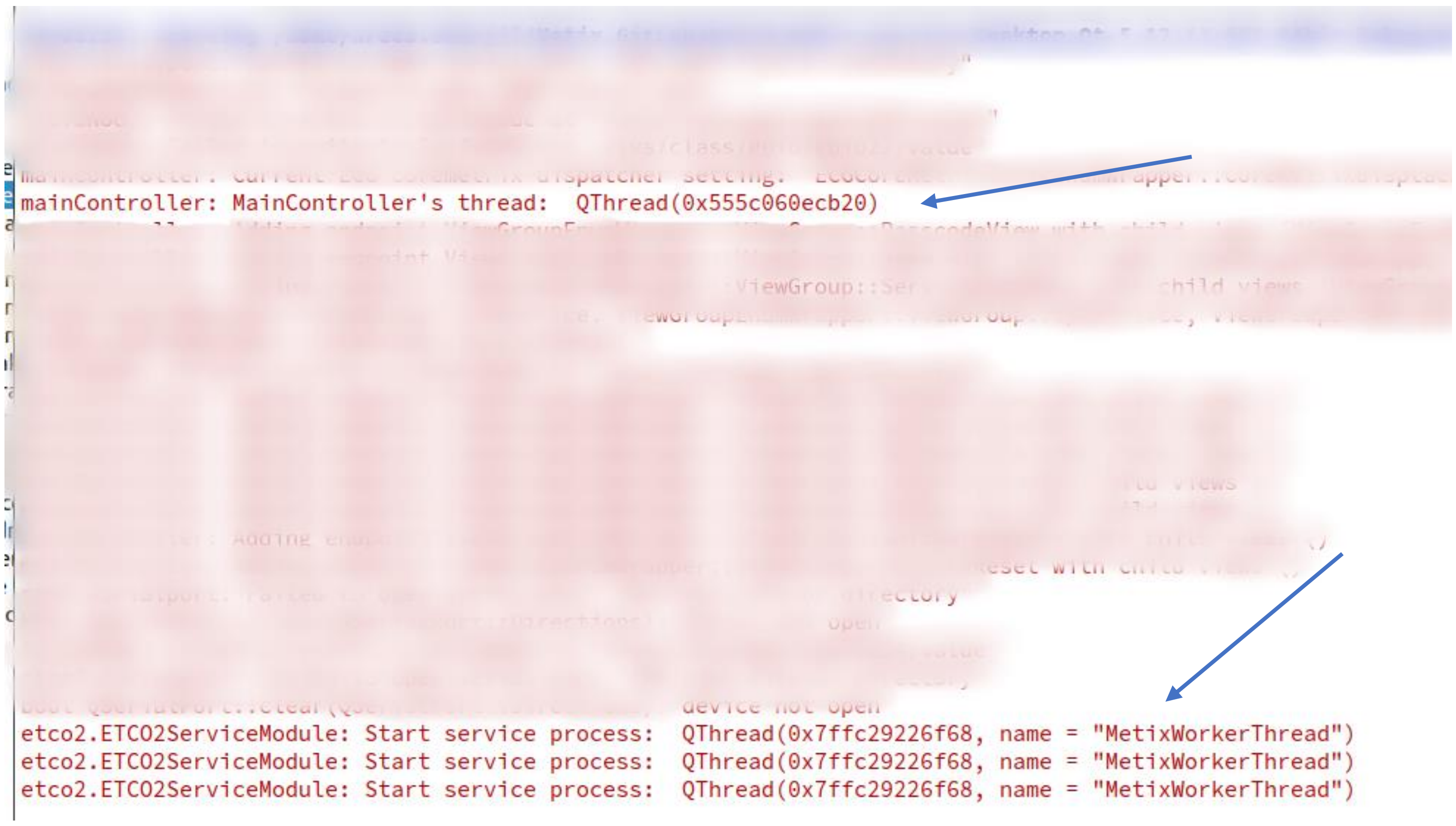
Figure 5.0. Analysing the debug logs for the GPS view. The top arrow shows the GUI thread while the bottom arrow shows the worker thread doing the GPS processing.

Referring to Listing 3.0, we can see the related classes for GPS processing were moved to a separate thread named as “WorkerThread”. The topmost arrow in Figure 5.0 prints out the thread where the MainController resides, this is the GUI thread and has no name set. The second arrow prints out the thread where the MetixNmeaPositionnfoSource and MetixNmeaSatelliteInfoSource reside. These two classes are responsible for processing the NMEA strings and displaying data to the GUI, as we previously discussed in the methodology section.



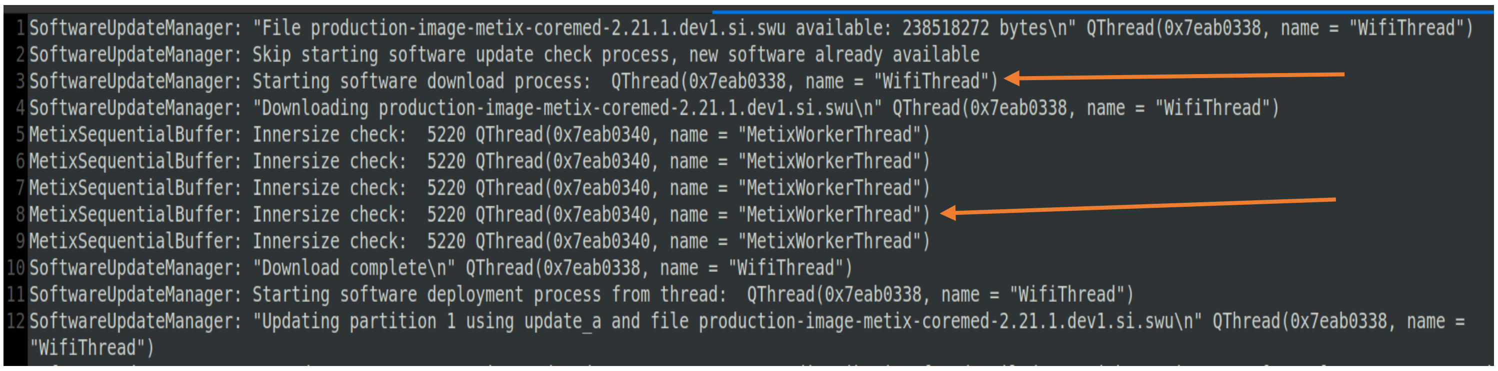
```
07:31:32: Starting /home/areeb.sherjil/[redacted]
mainController: MainController's thread:  QThread(0x564191217b20)
[redacted]
boot QSerialPort::clear(QSerialPort::Directions): device not open
NibpServiceModule: Starting calibration QThread(0x7fffc9d80898, name = "MetixWorkerThread")
NibpServiceModule: Attempting calibration of zero point
NibpServiceModule: Attempting calibration of span point
NibpServiceModule: Air leak test successful QThread(0x7fffc9d80898, name = "MetixWorkerThread")
```

Figure 5.1. Analysing debug logs from the NIBP service module. The top arrow shows the GUI thread while the bottom arrow shows the worker thread doing the NIBP processing.



```
[redacted]
mainController: MainController's thread:  QThread(0x555c060ecb20)
[redacted]
etco2.ETCO2ServiceModule: Start service process:  QThread(0x7ffc29226f68, name = "MetixWorkerThread")
etco2.ETCO2ServiceModule: Start service process:  QThread(0x7ffc29226f68, name = "MetixWorkerThread")
etco2.ETCO2ServiceModule: Start service process:  QThread(0x7ffc29226f68, name = "MetixWorkerThread")
```

Figure 5.2. Analysing debug logs from the CO2 service module. The top arrow shows the GUI thread while the bottom arrow shows the worker thread doing the CO2 processing.



```
SoftwareUpdateManager: "File production-image-metix-coremed-2.21.1.dev1.si.swu available: 238518272 bytes\n" QThread(0x7eab0338, name = "WifiThread")
SoftwareUpdateManager: Skip starting software update check process, new software already available
SoftwareUpdateManager: Starting software download process:  QThread(0x7eab0338, name = "WifiThread")
SoftwareUpdateManager: "Downloading production-image-metix-coremed-2.21.1.dev1.si.swu\n" QThread(0x7eab0338, name = "WifiThread")
MetixSequentialBuffer: Innersize check:  5220 QThread(0x7eab0340, name = "MetixWorkerThread")
MetixSequentialBuffer: Innersize check:  5220 QThread(0x7eab0340, name = "MetixWorkerThread")
MetixSequentialBuffer: Innersize check:  5220 QThread(0x7eab0340, name = "MetixWorkerThread")
MetixSequentialBuffer: Innersize check:  5220 QThread(0x7eab0340, name = "MetixWorkerThread")
MetixSequentialBuffer: Innersize check:  5220 QThread(0x7eab0340, name = "MetixWorkerThread")
SoftwareUpdateManager: "Download complete\n" QThread(0x7eab0338, name = "WifiThread")
SoftwareUpdateManager: Starting software deployment process from thread:  QThread(0x7eab0338, name = "WifiThread")
SoftwareUpdateManager: "Updating partition 1 using update_a and file production-image-metix-coremed-2.21.1.dev1.si.swu\n" QThread(0x7eab0338, name =
"WifiThread")
```

Figure 5.3. Analysing the debug logs from the SoftwareUpdateManager class during a software update. The top arrow shows the start of the download process and which thread it resides on. The bottom arrows show the size of the serial port data used for GPS processing on a separate thread.

The class MetixSequentialBuffer should be running on the MetixWorkerThread, refer to the UML diagram to are seen in action in the UML diagram in Figure 1.1. The bottom arrow in Figure 5.3 shows the periodic check on the serial port data. If the serial port data is unused after five seconds of build-up, the data is deleted. The two arrows confirm the software update process, and the GPS related processing are being done in separate threads. More information about the software update process can be found in appendix B.

Discussion

Implementing the GPS View

The GPS information worked as expected since that uses the Qt's QNmeaPositionInfoSource class. We know that Qt's classes would be tested thoroughly so we would not expect many sources of error here. The satellite information section displayed satellite information with the interval of five seconds. The signal status section turned out to be quite useful because it specifies the state of the signal. The satellites were in ascending order in signal strength.

The satellite information section did show some unexpected results. Some of the parameters namely signal strength did not match the latest NMEA string sent to the serial port. We saw this in table 4.0. Analysing the NMEA strings in Figure 4.3 shows that the satellites displayed in the GUI section are from the previous set of GSV strings and not the latest one. The satellite information is displayed using a five second timer.

The reason for the mismatch could be that if the timer times out and the slot executes, this can happen during the parsing of the NMEA strings. Whenever the timer slot executes it will emit the satellite information to the view class to display on the GUI even if latest NMEA strings are not processed. These could be processed and emitted in the next signal therefore no data is lost. If we analyse Figure 4.3 we can see that even in the previous NMEA strings the data in table 4.0(parameters that are highlighted red) still do not match. In this case we analyse the text file that contains all the NMEA strings used by the simulator app. The NMEA strings that match the GUI satellite information in Figure 4.2 are highlighted in yellow and that specific time is highlighted in green. This time is 9:48:58 pm which is four seconds behind the time shown in the latest NMEA strings in Figure 4.3 of 9:49:02 pm.

```
$GPGSV,3,1,11,15,63,177,21,24,59,260,27,23,39,276,17,13,39,132,32,0*68

$GPGSV,3,2,11,14,28,052,46,10,28,312,17,17,26,085,42,19,16,114,41,0*64

$GPGSV,3,3,11,12,15,201,24,21,06,010,20,44,02,107,,0*55

$GLGSV,3,1,09,65,67,284,22,74,67,050,35,75,52,232,28,72,46,174,24,1*71

$GLGSV,3,2,09,73,19,052,42,83,18,043,27,82,17,358,,66,15,326,,1*7C

$GLGSV,3,3,09,71,02,157,,1*46

$GNGLL,5601.3951,N,00349.4604,W,214857.000,A,A*5C

$GNRMC,214858.000,A,5601.3951,N,00349.4604,W,0.00,47.19,130822,,,A,V*2F
```

Listing 5.0. Finding the NMEA strings that match the data for Figure 4.2.

The analysis using the NMEA strings show that the satellite information section is not displaying the latest NMEA strings shown at the bottom of the NMEA simulator application. But we discussed the reason for this, and we can see that the satellite information shown in the GUI is from the previous set of NMEA strings. The timer interval could have something to do with this, perhaps with a shorter interval of say two seconds, this problem would not occur.

The satellite information algorithm lacked unit tests. To improve this, we should implement unit tests to automate the testing of satellite information outputted by our algorithm.

Multi-threading the service mode application

Multi-threading strategy 1.

“An alternative multi-threading approach, as offered by the QtThreadHelper repository on GitHub, proved to be a valuable solution, particularly for simpler multi-threading requirements where a single worker thread could be reused to execute functions away from the GUI thread. The key advantage of this approach was the seamless sharing of data between the GUI and the worker thread, facilitated by the implementation of a mutex within the library. This eliminated the need for developers to explicitly handle synchronization concerns, simplifying the programming process. However, a notable challenge arose when attempting to execute Qt's classes that emit signals on the worker thread, as accessing the source code of these classes was restricted. Additionally, creating the Qt object within the worker thread's exec(...) function proved to be time-consuming. These limitations, as discussed further in Appendix D, highlighted the complexities associated with integrating certain Qt objects into the worker thread, necessitating careful consideration when adopting this alternative approach” (ChatGPT, 2023) [20].

Multi-threading strategy 2.

The debug logs show that the two new threads WorkerThread and WifiThread are able to perform processing away from the GUI thread. This shows a successful strategy to implement parallel processing and improve performance of the device. The adoption of this approach also bolsters the scalability of the application. It ensures that the addition of new features in the future doesn't compromise the responsiveness of the graphical user interface (GUI). Utilizing multi-threaded software amplifies the system's ability to handle simultaneous tasks, leading to a significant boost in scalability. Such a structural decision provides an efficient and flexible platform for future software engineers. It furnishes them with a robust foundation, facilitating the process of building upon, refining, and augmenting the existing software to accommodate emerging needs and keep pace with the rapid technological advancements in the field.

The results of our study indicate positive outcomes. Specifically, our findings demonstrate that the processing of modules was successfully executed in the designated worker threads. Notably, during the thread migration process, it was observed that objects requiring their parent to be set to the object being moved exhibited the desired behaviour by residing on the target thread. As expected, when an object was moved to a thread, all its associated children were also successfully moved to the same thread. However, it should be noted that if a member variable did not have its parent correctly set to the object being moved to a different thread, that member variable did not reside on the intended thread. These observations highlight the importance of properly managing the parent-child relationships within the software system for seamless thread migration. “The successful implementation of multi-threading in the software system presents promising opportunities for enhanced performance and efficiency in the context of concurrent processing” (ChatGPT, 2023) [20].

Conclusions

The implementation of the GPS view in the service mode application using Qt's QNmeaPositionInfoSource class has shown promising results, demonstrating its usefulness and ease of use for processing GPS data. The introduction of the QNmeaSatelliteInfoSource class in Qt 6 holds potential for similar performance improvements. It would be valuable to compare the performance of the manual algorithm with Qt's class when the company transitions to Qt 6. Although the manual algorithm produced expected results, it exhibited a lag behind the latest NMEA strings due to the set interval of 5 seconds for satellite information updates. Implementing unit tests for the satellite processing algorithm is crucial to adhere to the company's test-driven development approach and ensure reliable results.

The multi-threading strategy employed using Qt's QThread object showcased good performance with the addition of two extra worker threads. However, challenges arose in protecting member variables using mutexes and correctly setting parent objects like timers. Notably, the ThreadHelper namespace provided by the company proved useful for asynchronous function execution. Debug logs illustrated the parallel processing occurring when users selected specific views. In future developments of the service mode application, careful consideration should be given to re-evaluating the optimal number of threads to ensure ideal performance.

“In conclusion, this dissertation has demonstrated advancements in the service mode application by leveraging Qt's GPS processing capabilities and implementing multi-threading techniques. The results highlight the potential of Qt's classes for efficient GPS data processing, while also emphasizing the importance of unit testing and considering performance implications when introducing new features" (ChatGPT, 2023) [20].

Appendix A: Brief overview of the Qt framework (C++).

In the world of microcontroller programming, signals in Qt can be thought of as interrupts and slots as interrupt handler functions.

To communicate between objects, the Qt framework uses the signal and slot mechanism. An example of event-driven communication would be if a user clicks the "update software" button we would want the software to begin updating. Other frameworks may use the callback technique to achieve this kind of communication. A callback is usually a function passed into another function as an argument which is then invoked from the outer function body.

The following diagram from Qt's documentation shows how the signal/slot mechanism work [17].

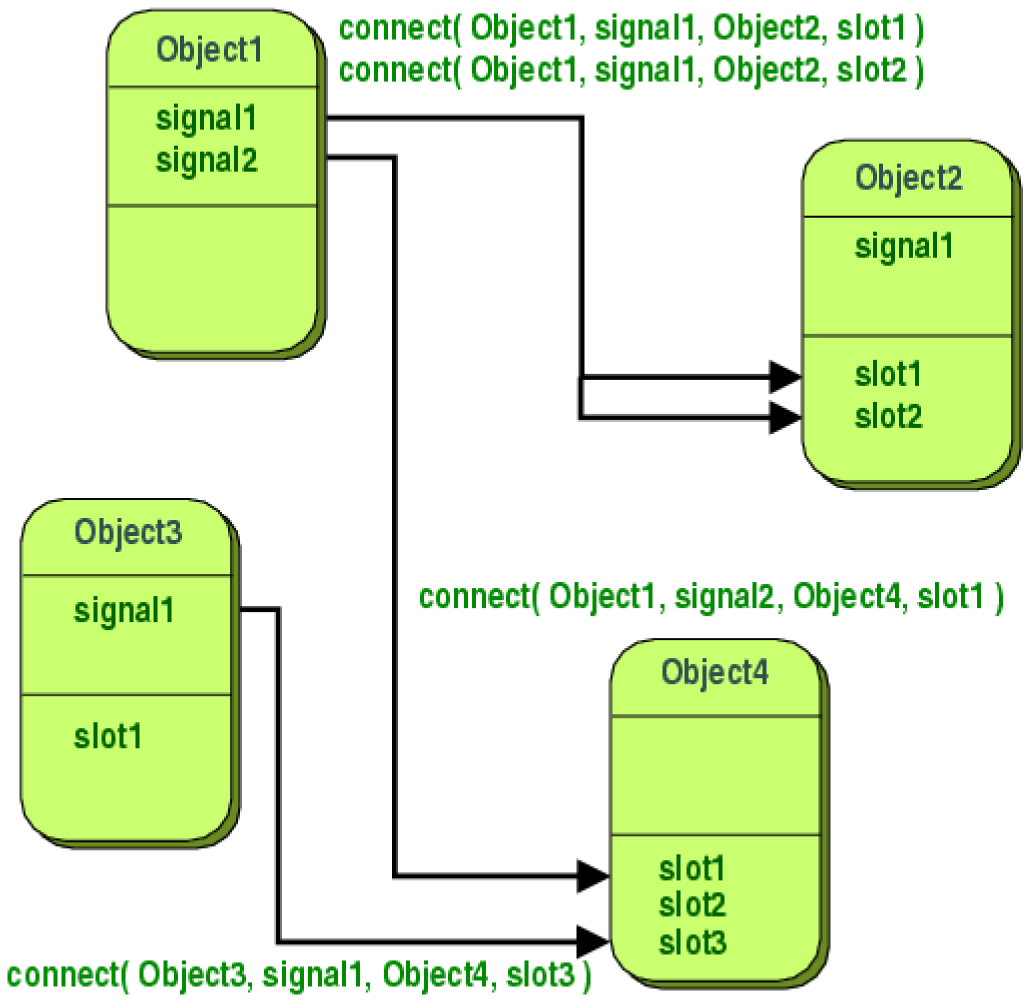

connect(Object1, signal1, Object2, slot1)
connect(Object1, signal1, Object2, slot2)
Object1
signal1
signal2
Object2
signal1
slot1
slot2
Object3
signal1
slot1
connect(Object1, signal2, Object4, slot1)
Object4
slot1
slot2
slot3
connect(Object3, signal1, Object4, slot3)

Figure A.0. Signal and slot mechanism visualized.

We can also use a simple code example to illustrate this point. Firstly, we create a signal, emit it and the slot would be executed in the corresponding object.

In situations where the two objects reside in separate threads we need to specify the connection type. After emitting the signal, we can choose for the slot to execute immediately(synchronously), or we can wait for the slot to execute in a separate thread(asynchronously). Qt contains enum types to make this possible, we can see this in figure 3.0. Qt::DirectConnection can be used for synchronous slot execution while Qt::QueuedConnection would invoke asynchronous execution.

```
void DummyNmeaSatelliteInfoSource::emitPendingSatelliteUpdate() {
    if (m_satellites.size()) {
        emit satellitesInViewUpdated(m_satellites.values());
        m_satellites.clear();
    }
}

void GPSServiceView::onShow() {

    // m_nmeaSatelliteSrc is of type DummyNmeaSatelliteInfoSource
    connect(&m_nmeaSatelliteSrc,
&DummyNmeaSatelliteInfoSource::satellitesInViewUpdated, this,
&GPSServiceView::receiveSatelliteinfo);

}

void GPSServiceView::receiveSatelliteinfo(const QList<QGeoSatelliteInfo>&
satellites) {
    // display the satellites provided in the function argument on the GUI
}
```

Figure A.1. Simple code example showing the signal/slot mechanism.

Here the class function emitPendingSatelliteUpdate() emits a signal "satellitesInViewUpdated", which is connected to the "receiveSatelliteInfo" in the GPSServiceView class. The class DummyNmeaSatelliteInfoSource processes GPS data and sends that to GPSServiceView class which is responsible for displaying that to the user. In this example DummyNmeaSatelliteInfoSource would be the emitter and GPSServiceView would be receiver.

In the latter part of the dissertation, we know that DummyNmeaSatelliteInoSource class was moved to a separate thread (see Listing 3.0) to ensure that GPS processing is not computed on the GIU thread. In this case we require an asynchronous computation as we need the slot to execute in the receiver's thread which would be the GUI thread. Now we specify the connection type as Qt::QueuedConnection.

```
void GPSServiceView::onShow() {
    // m_metixnmeaSatelliteSrc is of type MetixNmeaSatelliteInfoSource
    connect(&m_nmeaSatelliteSrc,
&DummyNmeaSatelliteInfoSource::satellitesInViewUpdated,
        this, &GPSServiceView::receiveSatelliteinfo, Qt::QueuedConnection);
}
```

Figure A.2. Specifying the connection type in when the emitter and receiver reside in separate threads.

Appendix B: Overview of the service mode application.

The MainController is large class responsible for performing navigation on the service mode application and dealing with the different device modules. It is not possible to show all the functions and private member variables otherwise the UML class diagram would become difficult to manage. Here is a simplified class diagram with the un-needed details removed:

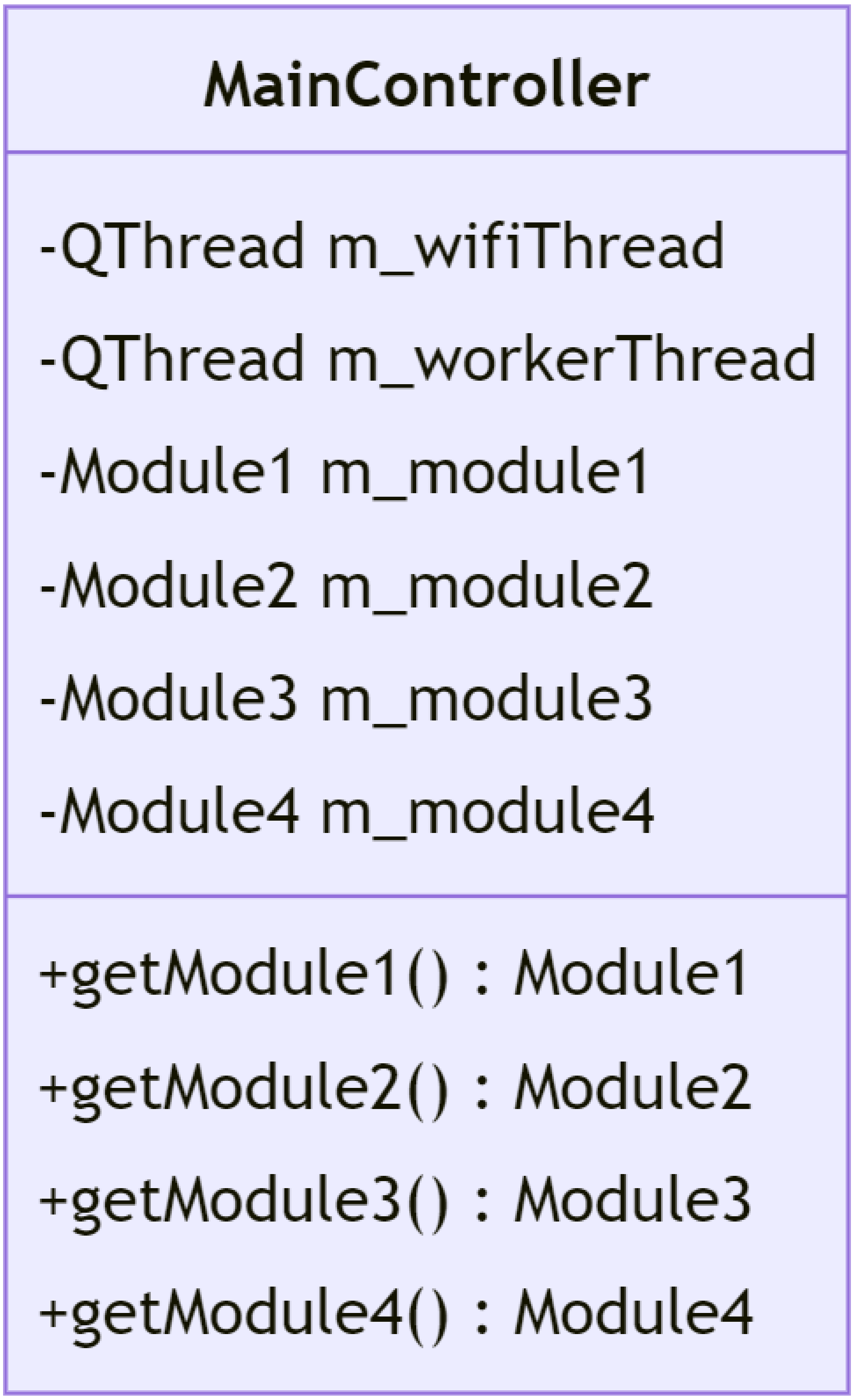


Figure B.0. UML class diagram for the MainController.

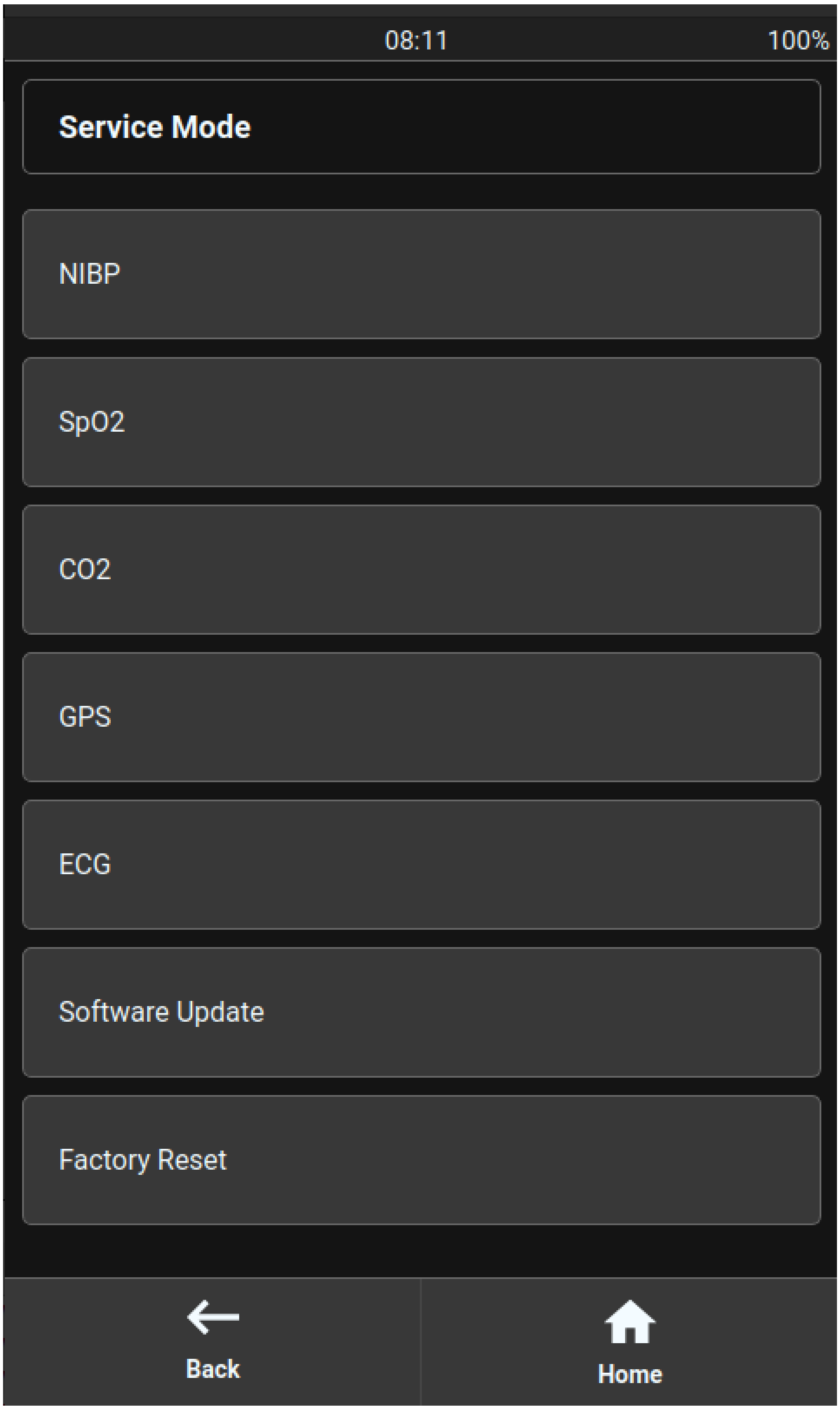
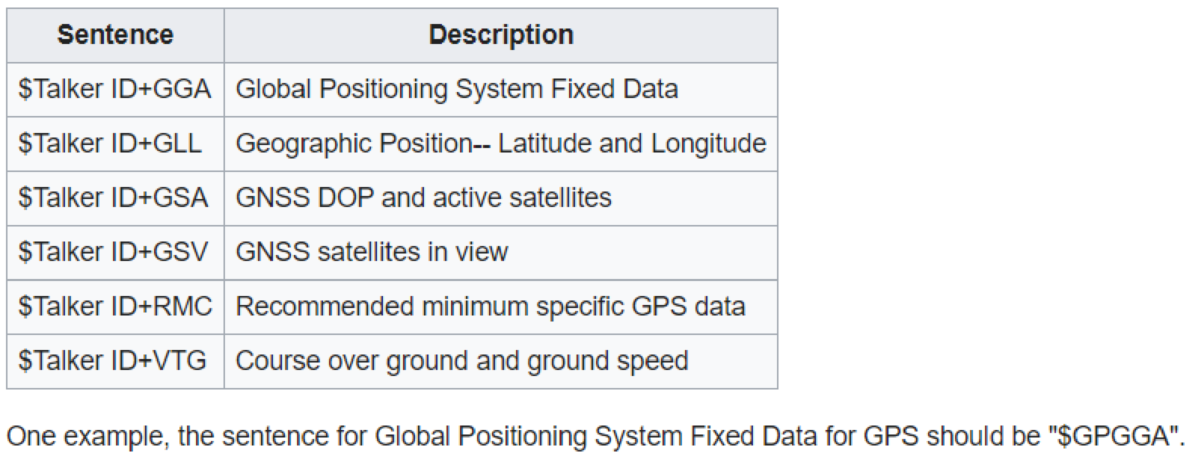


Figure B.1. Screenshot of the service mode application views.

The service mode contains different views for parameters such as shown above. The user can choose to investigate and or debug a certain parameter like GPS. All of the views perform certain computations and are required to be moved on a separate thread.

Appendix C: Implementing the GPS view.

The device receives GPS information through a serial port as NMEA strings. NMEA stands for National Marine Electronics Association, it is a standard method of communication for GPS receivers. There are many different types of NMEA strings, each one of them provide different position information.



Sentence	Description
$Talker ID+GGA	Global Positioning System Fixed Data
$Talker ID+GLL	Geographic Position-- Latitude and Longitude
$Talker ID+GSA	GNSS DOP and active satellites
$Talker ID+GSV	GNSS satellites in view
$Talker ID+RMC	Recommended minimum specific GPS data
$Talker ID+VTG	Course over ground and ground speed

One example, the sentence for Global Positioning System Fixed Data for GPS should be "$GPGGA".

Figure C.0. Different types of NMEA strings [18].

Displaying the GPS positional information was done using Qt's class QNmeaPositionInfoSource [2]. The time to first fix (TTFF) was the time it took for the first valid position signal appear after the device boot. The boot time of the linux device was extracted from the sysinfo struct [19].

```
DummyNmeaPositionInfoSource::DummyNmeaPositionInfoSource(QObject* parent, bool 
computeTTFF)
    : m_nmeaSrc(QNmeaPositionInfoSource::RealTimeMode, this) {
    connect(&m_nmeaSrc, &QNmeaPositionInfoSource::positionUpdated,
        this, &AbstractNmeaPositionInfoSource::positionUpdated);
    connect(&m_nmeaSrc, &QNmeaPositionInfoSource::updateTimeout,
        this, &AbstractNmeaPositionInfoSource::updateTimeout);

    if (computeTTFF) {
        connect(this, &DummyNmeaPositionInfoSource::positionUpdated,
            this, &DummyNmeaPositionInfoSource::estimateTTFF);
    }
}

void DummyNmeaPositionInfoSource::estimateTTFF(const QGeoPositionInfo& update) {
    if (update.isValid()) {
        qCDebug(log) << "Computing TTFF";
        struct sysinfo si;
        sysinfo(&si);
        m_TTFF = si.uptime;
        emit estimatedTTFFUpdated(m_TTFF);
        disconnect(this, &DummyNmeaPositionInfoSource::positionUpdated,
            this, &DummyNmeaPositionInfoSource::estimateTTFF);
    }
}
```

Listing C.1. Code example that computes the estimated time to first fix.

The signal originating from DummyNmeaPositionInfoSource is channelled to a slot designed to extract and exhibit the GPS data on the graphical user interface (GUI). In this process, we strategically employ the 'float' data type instead of 'double'. This decision is grounded in our objective to minimize memory consumption, given that the 'double' type requires twice the memory capacity compared to 'float'. Therefore, our approach effectively optimizes memory usage without compromising the accuracy and functionality of the GPS information display.

```
GPSServiceView::GPSServiceView() {
    // other code to be ignored...
    // we make the connection here:
    connect(&m_nmeaSrc, &DummyNmeaPositionInfoSource::positionUpdated,
        this, &GPSServiceView::receiveGpsPosUpdate);
}

void GPSServiceView::receiveGpsPosUpdate(const QGeoPositionInfo& update) {

    if (update.isValid() == false) {
        qWarning(log) << "Nmea Error Code= " << m_metixnmeaSrc.error();

        for (GPSInfoIndex index : k_infoStringMap.keys()) {
            if (index != GPSInfoIndex::SignalStatus) {
                m_gpsinfocontainer->updateValue(k_infoStringMap.value(index),
                    "--");
            }
        }
    }
    else if (update.isValid() == true) {

        m_gpserrorTimer.start(k_nmeaUpdateTimeoutTime);// restart the error timer

        // here we extract GPS information parameters from the emitted signal:

        float l_latitude = update.coordinate().latitude();
        char  latitude_hemisphere = (l_latitude < 0) ? 'S' : 'N';
        float l_longitude = update.coordinate().longitude();
        char  longitude_hemisphere = (l_longitude < 0) ? 'W' : 'E';

        float l_altitude = update.coordinate().altitude();
        float speed = update.attribute(QGeoPositionInfo::GroundSpeed);

        float horizontal_accuracy = update.attribute(QGeoPositionInfo::HorizontalAc-
curacy);
        float vertical_accuracy = update.attribute(QGeoPositionInfo::VerticalAccu-
racy);
        float positionaccuracy = sqrt(horizontal_accuracy * horizontal_accuracy
            + vertical_accuracy * vertical_accuracy);

        float bearing = update.attribute(QGeoPositionInfo::Direction);

        // add the necessary parameters to the display...
    }
}
```

Listing C.2. Extracting and displaying the GPS information to the GUI.

“Within the constructor, we observe the signal being connected to the slot within the GPSServiceView class. This slot is responsible for extracting GPS information from the supplied function argument, which is of the QGeoPositionInfo type [5]. Notably, the error timer undergoes a reset with each valid signal detection. However, in the absence of a valid signal and subsequent to a one-second timeout, the GUI defaults to displaying the signal status as "inactive." This dynamic representation of signal status on the GUI provides real-time feedback and enhances system responsiveness” (ChatGPT, 2023) [20].

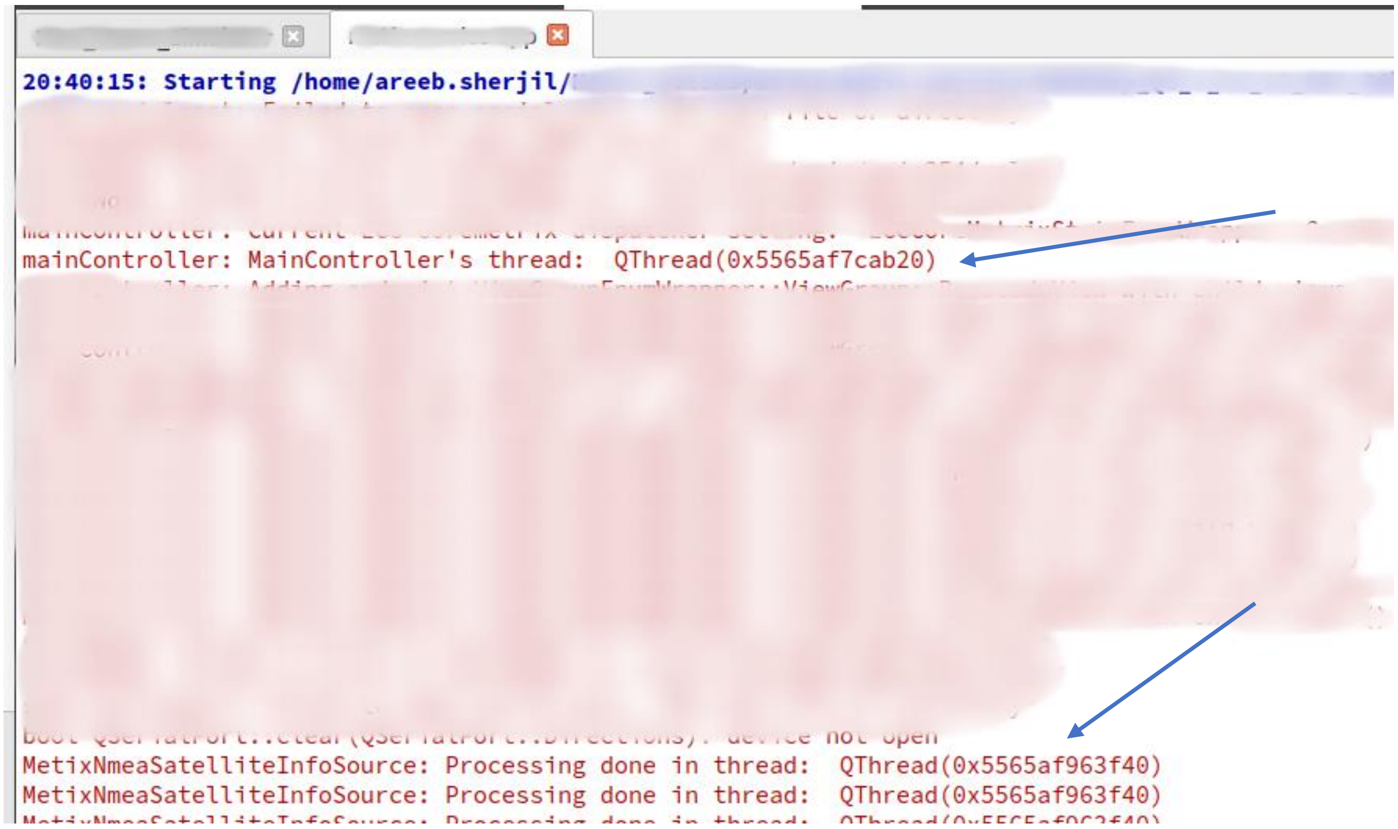
Appendix D: Implementing QtThreadHelper multi-threading strategy.

This multi-threading strategy was adapted from the QtThreadHelper GitHub repository [13]. This repository is designed to help with situations where we want to re-use the same thread for multiple tasks.

This strategy is deployed using a WorkerThread object. To run the processing of separate views on the worker thread, the thread object had to be passed into that class. A pointer to the thread object was passed into different classes. The necessary functions were ran inside the lambda function of the exec(…) function. Debug logs were used to check if the functions were running in the different thread.

```
// MainController.h------------------

class MainController : public QObject {
    Q_OBJECT
public:
    // other code to be ignored...
private:

    /// Worker thread for the GPS module
    std::unique_ptr<WorkerThread> m_workerThread;
};

// MainController.cpp------------------

MainController::MainController()
    : m_workerThread{ std::make_unique<WorkerThread>() },
    m_serialDemux{ ConfigurationSettingsHelper::readSetting("PORTS", "gpsPort-
Name"), m_workerThread.get() },
    m_nmeaGps{ this, true, m_workerThread.get() },
    m_nmeaSatellites{ MetixNmeaSatelliteInfoSource::UpdateMode::RealTimeMode,
m_workerThread.get() } {

    // other code to be ignored...
    qCDebug(log) << "MainController's thread: " << QThread::currentThread();
}

// MetixNmeaSatelliteInfoSource---------------

MetixNmeaSatelliteInfoSource::MetixNmeaSatelliteInfoSource(MetixNmeaSatelliteIn-
foSource::UpdateMode mode,
    WorkerThread* workerThread)
    :m_updateTimer{ this }, m_updateMode{ mode }, m_workerThread{ workerThread }
{

    connect(&m_updateTimer, &QTimer::timeout, this,
        &MetixNmeaSatelliteInfoSource::emitPendingSatelliteUpdate);

    qRegisterMetaType<QList<QGeoSatelliteInfo>>("QList<QGeoSatelliteInfo>");
}


void MetixNmeaSatelliteInfoSource::parsefromDemux() {
    m_workerThread->exec([this]() {
        qCDebug(log) << "Processing done in thread: " << QThread::cur-
rentThread();
        if (m_device->canReadLine()) {
            while (m_device->canReadLine()) {
                char buf[1024];
                int len = m_device->readLine(buf, sizeof(buf));
                processNmeaData(buf, len);
            }
        }
        });
}
```

Listing D.0. Creating a pointer to worker thread and passing that into MetixNmeaSatelliteInfoSource class.



```
20:40:15: Starting /home/areeb.sherjil/[redacted]
[redacted]
mainController: MainController's thread:  QThread(0x5565af7cab20)
[redacted]
MetixNmeaSatelliteInfoSource: Processing done in thread:  QThread(0x5565af963f40)
MetixNmeaSatelliteInfoSource: Processing done in thread:  QThread(0x5565af963f40)
```

Figure D.0. Analysing debug logs from the Listing D.0.

The code example in Listing D.0 shows that we create a pointer to WorkerThread using a smart pointer (std::unique_ptr). The raw pointer is then passed into the different classes that need to run functions on a separate thread. Inside MetixNmeaSatelliteInfoSource, we make use of the exec(…) function to run the function body in the worker thread. We can see from the debug logs (Figure D.0) that the MainController thread (the main or GUI thread) is different from the worker thread.

The problem with this approach of using QtThreadHelper was that in order to run an object on the worker thread, we either need to pass that function to the exec(…) function like shown in Listing D.0 or we create that object inside the worker thread's exec(…) function. This is problematic because the class MetixNmeaPositionInfoSouce class contains QNmeaPositionInfoSource. The MetixNmeaPositioInfoSource class is a wrapper around the Qt's class. Because of the difficulty and the need for refactoring the existing codebase this strategy was abandoned.

This repository can be implemented for simple situations that require multi-threading. In our case however, we do not find this to be suitable. In the service mode we require the software to be scalable so it can be expanded upon in the future.

References

[1] “multithreading - What is a ‘thread’ (really)?,” Stack Overflow. https://stackoverflow.com/questions/5201852/what-is-a-thread-really

[2] “QNmeaPositionInfoSource Class | Qt Positioning 6.5.0,” doc.qt.io. https://doc.qt.io/qt-6/qnmeapositioninfosource.html (accessed May 21, 2023).

[3] Fred Zahradnik, “What Does Time to First Fix (TTFF) Mean?,” Lifewire. https://www.lifewire.com/time-to-first-fix-ttff-1683313 (accessed May 21, 2023).

[4] N. Couronneau, P. J. Duffett-Smith, and A. Mitelman, “Calculating Time-to-First-Fix,” GPS World, Nov. 02, 2011. https://www.gpsworld.com/wirelessinfrastructurecalculating-time-first-fix-12258/

[5] Qt Framework, “QGeoPositionInfo Class | Qt Positioning 6.5.0,” doc.qt.io. https://doc.qt.io/qt-6/qgeopositioninfo.html (accessed May 21, 2023).

[6] Devendranaga, “NMEA checksum calculator in C,” Gist, Jan. 01, 2016. https://gist.github.com/devendranaga/fce8e166f4335fa777650493cb9246db (accessed May 21, 2023).

[7] “Container Classes | Qt Core 6.5.1,” doc.qt.io. https://doc.qt.io/qt-6/containers.html (accessed Jun. 24, 2023).

[8] Petrbel, “C++ std::unordered_map complexity,” Stack Overflow, Jan. 01, 2014. https://stackoverflow.com/questions/19610457/c-stdunordered-map-complexity (accessed May 21, 2023).

[9] Q. Framework, “QSet Class | Qt Core 6.5.0,” doc.qt.io. https://doc.qt.io/qt-6/qset.html (accessed May 21, 2023).

[10] Allthings.how, 2023. https://allthings.how/content/images/wordpress/2022/05/allthings.how-how-to-use-winrar-on-windows-11-image-13.png (accessed May 21, 2023).

[11] “Multithreading Technologies in Qt | Qt 6.5,” doc.qt.io. https://doc.qt.io/qt-6/threads-technologies.html (accessed May 21, 2023).

[12] T. S., “New threads vs. reusing threads,” Stack Overflow, Jul. 01, 2015. https://stackoverflow.com/questions/24361779/new-threads-vs-reusing-threads (accessed May 21, 2023).

[13] S. Schröder, “QtThreadHelper,” GitHub, Apr. 03, 2023. https://github.com/SimonSchroeder/QtThreadHelper (accessed May 21, 2023).

[14] krzych, “Why using QMetaObject::invokeMethod when executing method from thread,” Stack Overflow, Jan. 01, 2013. https://stackoverflow.com/questions/13948337/why-using-qmetaobjectinvokemethod-when-executing-method-from-thread (accessed May 21, 2023).

[15] “Qt Namespace | Qt Core 6.5.0,” doc.qt.io. https://doc.qt.io/qt-6/qt.html (accessed May 21, 2023).

[16] “QMetaObject Struct | Qt Core 6.5.0,” doc.qt.io. https://doc.qt.io/qt-6/qmetaobject.html#invokeMethod-8 (accessed May 21, 2023).

[17] “Signals & Slots | Qt Core 6.2.4,” doc.qt.io. https://doc.qt.io/qt-6/signalsandslots.html

[18] T. B. Shields, “Introduction to GPS Data NMEA & RTCM Donald Choi, ALS/G2. - [PPT Powerpoint],” fdocuments.in. https://fdocuments.in/document/introduction-to-gps-data-nmea-rtcm-donald-choi-alsg2.html (accessed May 21, 2023).

[19] “sysinfo(2) - Linux manual page,” man7.org. https://man7.org/linux/man-pages/man2/sysinfo.2.html (accessed May 21, 2023).

[20] OpenAI, “ChatGPT,” chat.openai.com, 2023. https://chat.openai.com/