Design of a Graphical User Interface for Unmanned Aerial Vehicles

Richard Perrin

University of New South Wales at the Australian Defence Force Academy

Unmanned Aerial Vehicle (UAV) technology has seen significant research in the last decade: smaller, lighter systems have been developed with increased capability and range. Research into autonomous flight is conducted at UNSW Canberra with miniature unmanned helicopters, primarily the Hirobo Eagle, Yamaha RMAX and the Vario XLC. These aircraft have been fitted with flight computers that enable autonomous flight. Unfortunately the helicopters have a very low endurance of less than an hour, where the Eagle only has a maximum flight time of fifteen minutes. Thus it is important to maximise the productivity of each testing session. A Graphical User Interface (GUI) run from a ground computer, wirelessly connected to the helicopter, allows the flight computer software to be remotely modified during flight. This paper outlines the development of a Microsoft-Windows-compatible GUI that will replace the Real-Time Linux version, providing not only equivalent functionality, but a number of other features and improvements.

Contents

I. Project Background 2
   A. Existing Linux GUI 2
   B. Research and industry examples of UAV GUIs 3
II. Project Aims 3
III. Software Development Process 3
   A. Analysis 4
   B. Design 4
      1. Conceptual Model 4
      2. Selection of Software Language 5
         a. External Library Requirements 5
         b. Ease of Use 5
         c. Code Efficiency 6
         d. Online Information and Support 6
         e. Software Language Comparison Summary 6
   C. Major Software Stages 7
      1. Separation of Command Frames in a Stream 7
      2. Integration of UAV Command Functionality 7
      3. Improvement of the parsing routine 8
      4. Numerical Display of the UAV Telemetry Feed 9
      5. Incorporation of Virtual Instruments 9
   D. Other Improvements 10
      1. Settings tab 10
      2. Automatic Settings Configuration 11
IV. Creation of GUI Adaptation Instructions 11
V. Conclusions 11
VI. Recommendations 11

Acknowledgements 11
References 11

APPENDICES
Appendix A. Project Management Plan A1
Appendix B. Command Functionality Test Sheet

I. Project Background

UNSW Canberra has three primary helicopters with which research into UAV technology is conducted: the Yamaha RMAX, the Vario XLC and the Hirobo Eagle. The helicopters are used for research into unmanned flight, including high bandwidth control and visual flight control. All helicopters implement a flight computer with a two-way telemetry protocol to a ground control station. The hardware used in each flight computer varies with the helicopter type. A ground computer running a Real-Time Linux (RTLinux) based GUI allows parameters in the flight computer’s internal control algorithms and servo settings to be altered. During flights, data is logged and can be analysed to improve the control system. The flight computers and the GUI were designed by Dr Garratt in his research on biologically inspired vision and control of UAVs (Garratt, 2007). Unfortunately, the GUI is only compatible with the legacy RTLinux Operating System (OS), which, aside from its age, is becoming increasingly difficult to find. Since Windows-based computers are readily available, it has become a research priority to develop a replacement GUI for the Windows OS.

The GUI communicates with UAVs over a number of serial data carrier formats, including Wi-Fi and Bluetooth. Communication is in the form of command frames; a series of bytes that have a consistent format, shown below in Figure 1

<table>
<thead>
<tr>
<th>Byte Number</th>
<th>Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Header (the decimal value 85 represented in binary)</td>
</tr>
<tr>
<td>1</td>
<td>Command Code</td>
</tr>
<tr>
<td>2</td>
<td>Number of final byte: n</td>
</tr>
<tr>
<td>3</td>
<td>Identity number of command, i.e. the number of commands that have preceded the current command. The ID is represented as MSB and LSB.</td>
</tr>
<tr>
<td>4</td>
<td>Command parameters; commands can require a single parameter, multiple, or none at all.</td>
</tr>
<tr>
<td>5 → n-1</td>
<td>Checksum; the sum of all byte values from byte 1 to n-1. If the value overflows one byte, the checksum is equal to the remainder.</td>
</tr>
</tbody>
</table>

Figure 1. GUI communication format.

Numeric command parameters are converted to byte form in accordance with IEEE Std 754. The GUI awaits a receipt from the UAV, which is the exact same command frame that was sent, to confirm that the data has been received. The command will be resent to the UAV until this occurs.

A. Existing Linux GUI

The existing software (Figure 2) is run from a ground station computer running Real-Time Linux. The GUI allows users to change parameters in the UAV’s internal control algorithms and servo settings with the benefit that the UAV flight computer can be updated midflight, allowing more testing to be done in a single session. Simultaneously, the UAV sends telemetry information to the ground computer, which is displayed in separate windows either numerically or graphically. A test computer with the same onboard software as the Eagle UAV has been used for testing of the GUI which has communication via a direct (wired) RS-232 connection.

Figure 2. Existing Linux GUI.
B. Research and Industry Examples of UAV GUIs

Commercially-available, Windows-compatible UAV GUIs and examples from other research have been studied to determine benchmarks for functionality, appearance and additional features. A previous project at UNSW Canberra produced a Personal Digital Assistant (PDA) GUI that replicated some of the functionality of the existing Linux GUI, demonstrating the advantages of mobility of the ground computer (Kaberry, 2006).

MicroPilot, a manufacturer of miniature UAV Autopilots has produced a GUI for its autopilots called HORIZON, shown in Figure 3. HORIZON has a GUI that integrates maps, video feed and virtual instrument data. HORIZON connects to UAVs over a radio-frequency link from an external radio modem. HORIZON allows users to create their own command buttons, similar to the aims of this project of future adaptability of the software (www.micropilot.com).

Cloud Cap Technology produces a GUI for their UAVs called the Piccolo Command Centre (PCC). The PCC GUI has a similar interface to HORIZON, offering a central 3D map with virtual instruments, flight data feeds and a choice of additional windows, similar to the widgets that will be designed for this project. The PCC must be running on a computer connected to the Cloud Cap ground station which then connects wirelessly to the UAVs (www.cloudcaptech.com).

Despite the available software, a unique solution is required for this project. The commercial GUI software described previously is only compatible with selected UAVs, produced by the same company. Furthermore, this project requires an open source solution: further research into UAVs will require changes to the software. Finally, the UNSW Canberra UAVs use a customised communications stack, meaning that even if an open source commercial solution was found, it would almost certainly require a complete overhaul.

II. Project Aims

This project had three aims, which formed the project specific deliverables:

1. The development of software capable of a reliable telemetry/command-and-control link between a laptop computer running windows software and a UNSW Canberra UAV. Software should be compatible with the existing communications stack running on these vehicles and be configurable to either serial (RS232) /Wi-Fi or Bluetooth carriers;
2. The design and implementation of a Windows compatible GUI that provides the same control functions and display capabilities of the existing real-time Linux based GUI;
3. To write a ‘HOW-TO’ document with example source code for future users to help them develop extensions to the software.

III. Software Development Process

The Staged Delivery software development process, also known as Rapid Application Development, has been adopted for this project. The steps in this process are shown in Figure 4. In the staged delivery method, the developer initially decides on the requirements and functionality of the project end state, and then a series of stages to achieve it. The advantage of this process is that major design decisions can be made in the conceptual phase, whilst minor details can be finalised at the appropriate stage. This leaves more time for software
development, whilst providing the developer with the necessary design outline. The staged delivery method is an industry standard software design process for GUI development (Dorsey, 1996). It is the most effective design process for this project for the following reasons:

- The end state of the project is clear, thus intermediate stages can be easily conceptualised.
- The complete system is too complex and would carry too much risk to create in one stage.
- Communication capabilities can be developed and tested rapidly before proceeding to subsequent graphical development.
- Tangible signs of progress can be provided to the client with each software stage.

![Figure 4. Staged Delivery Method](image)

This project was developed in accordance with J-STD-016-1995 which supersedes IEEE standard 1498 and defines a set of software development activities throughout the design process. The requirements of the standard include a systematic documented method of software development and traceability between levels of requirements and design stages. Development activities for user-focused-software, outlined in ISO 9241-210, were applied throughout the project. These include user requirements analysis, user-focused-design, and periodic user evaluation of the software. By following both the staged delivery method, and the guidelines set by these standards, there has been a clear work schedule and the project has not suffered from any architectural failures.

**A. Analysis**

The end state of this project was a windows compatible GUI capable of all of the functionality of the existing Linux UAV GUI, adaptable to future requirements. The capabilities of the existing GUI and the specific windows compatibility requirement formed the basis of the user requirements and thus the project design requirements:

1. Graphical/numerical representation of UAV autopilot parameters
2. Control of UAV autopilot parameters
3. Graphical/numerical representation of UAV sensor data.
4. Communication with the UAV over either a serial RS-232 or a TCP/IP connection
5. Windows compatibility
6. Adaptability to future requirements

A GUI that fulfils these requirements will satisfy the end state of the project.

**B. Design**

1. **Conceptual Model**

   The conceptual model of the software architecture is shown in Figure 5, demonstrating the proposed flow of information. Players (Software Objects) and Agents (Coded Processes) are shown to describe the internal interactions of the program.
2. Selection of Software Language

The most crucial and irreversible decision that was made during the conceptual design phase was the selection of the programming language. As a result, a detailed evaluation was conducted to determine the most appropriate language to use.

Since there are hundreds of programming languages in existence, many of which are quite similar, only the major languages were considered. Less common, yet unique languages were not considered, since lack of online information would be of significant detriment to future adaptability. The selection was limited to languages that are realistically capable of fulfilling each design requirement: Java, C++, C#, Visual Basic (VB.NET), and MATLAB. MATLAB is not a separate language, however it was included as it is a tool regularly used by engineers.

These options were compared based on four factors; external library requirements, ease of use, code efficiency and the online information/support. Typically, program execution speed would be examined when deciding on a programming language; however this was not the case for this project. The communication rate was not deemed critical, as UAV control is not real-time, and all languages considered here are capable of creating an executable that can operate at sufficient speeds for the required functionality.

2.1. External Library Requirements

Most languages are fully equipped to write programs requiring basic functionality, such as calculations and data processing. However, graphical display and TCP/IP or serial communications are usually not standard features, requiring libraries to be imported into the program. In some cases, add-ons are required to be downloaded or even bought. The most desired language was the one where no external libraries or classes needed to be downloaded, however, languages which require minimal import statements were considered acceptable, since only minimal lines of additional code would be required. After some research, it was found that all languages with the exception of MATLAB require a number of software libraries to be imported; MATLAB requires the purchasing of the GUI Developer Environment to meet the graphical display requirements of this project (http://www.mathworks.com.au).

2.2. Ease-of-Use

The ease-of-use of each language was an important consideration, since the final design requirement specifies an adaptable program. This will be best achieved with the most easy-to-use language. Languages were compared based on the readability of the code, and how intuitive the language is to use. Specific design requirements were also considered such as ease of graphical implementation. Unfortunately, ‘Ease-of-use’ is a subjective quantity which has received limited research, however, a study in 1997 (Jones, C) related the ease-of-use of a language to the productivity average of novices using it. Language levels were then assigned for increased productivity, the higher the better. An excerpt of the results is shown in Figure 6.

<table>
<thead>
<tr>
<th>Language</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++</td>
<td>6.00</td>
</tr>
<tr>
<td>Fortran</td>
<td>4.5</td>
</tr>
<tr>
<td>Java</td>
<td>6.00</td>
</tr>
<tr>
<td>Machine Code</td>
<td>0.5</td>
</tr>
<tr>
<td>Visual Basic</td>
<td>11.00</td>
</tr>
</tbody>
</table>

Figure 6. Language ease-of-use levels
Neither MATLAB nor C# were analysed, however, C# was assumed to be similar to Java for ease-of-use, given that it is designed to compete with it at a commercial level. MATLAB on the other hand, despite being based on C++, Fortran and Java, was considered easier to use than all three. Experience with both MATLAB and visual basic suggested that MATLAB would be easier to use for solving simple problems, however the ease of GUI creation was unknown.

c. Code Efficiency

Since this project has a short timeframe, the programming language which is most efficient to develop with was desired. There are several measures of efficiency, for example, a comparison of time required to solve a given problem with various languages. However for a less experienced user, the number of code lines required is also a useful measure. The Karlsruhe institute of technology conducted a study of a variety of major languages for both of these measures (Prechelt et al, 2000, p6) as shown in Figure 7:

![Figure 7. Code efficiency measurements](image)

Though C#, Visual Basic and MATLAB were not studied, it was assumed that C# is comparable to Java and C++, since it is a semi-compiled language with roots in C. Visual Basic and MATLAB were compared to Python as they are interpretive languages, and neither are known to be code intensive. Furthermore, GUI creation is very straightforward in visual basic, command windows and buttons are simply dragged into the project view with minimal code necessary. From this data we can see that the languages that would take longest to solve a problem are Java and C++, and implicitly, C#. It is more likely that MATLAB and Visual basic could be used to solve a problem with less coding time and length.

d. Online Information and Support

When developing software, the availability of online information and support is critical to success, particularly for a lone developer, as in the case of this project. There are few studies on online support for programming languages, and such studies would become quickly redundant. However, measuring online references to specific languages provides a useful, up-to-date indication of online information available. A list of the most referenced software languages online, released in May 2012 (Figure 8), showed that, of the languages under consideration, Java received the most references, followed by C++, C# and the Visual Basic. MATLAB was not listed (www.tiobe.com).

<table>
<thead>
<tr>
<th>Rank</th>
<th>Language</th>
<th>% total references</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>C</td>
<td>17.346%</td>
</tr>
<tr>
<td>2</td>
<td>Java</td>
<td>16.599%</td>
</tr>
<tr>
<td>3</td>
<td>C++</td>
<td>9.825%</td>
</tr>
<tr>
<td>4</td>
<td>Objective-C</td>
<td>8.309%</td>
</tr>
<tr>
<td>5</td>
<td>C#</td>
<td>6.823%</td>
</tr>
<tr>
<td>6</td>
<td>PHP</td>
<td>5.711%</td>
</tr>
<tr>
<td>7</td>
<td>Visual Basic</td>
<td>5.457%</td>
</tr>
</tbody>
</table>
e. Software Language Comparison Summary

Each language was scored for its performance in each area based on available information. The best performing language in each area was given a 5. Subsequently lower scores were awarded based on the level of disparity between the next best language in that area. The results are shown in Figure 9, from this table it was evident that the most appropriate language to use for this project was VB.NET

<table>
<thead>
<tr>
<th>Language</th>
<th>Java</th>
<th>C++</th>
<th>C#</th>
<th>VB.NET</th>
<th>MATLAB</th>
</tr>
</thead>
<tbody>
<tr>
<td>External Library Require</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>Ease of Use</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Code Efficiency</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Online Support</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Total</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>17</td>
<td>13</td>
</tr>
</tbody>
</table>

Figure 9. Software Language Comparison Summary

C. Major Software Stages

After the conceptual design was finalised, the project was separated into software stages, each with a clear and precise function that would be tested at its completion. The delivery schedule is shown in the project management plan in Appendix A. It would be impractical to detail the accomplishment of each stage; instead, this section will describe the major, defining stages in the development process.

1. Separation of Command Frames in a Stream (Stage 7)

In addition to sending command receipts, the UAV sends telemetry information from its different sensors to the ground computer at a rate of 50Hz. As a result, one of the most important steps in the development process was to design an efficient, robust incoming data parsing procedure for the GUI. The parsing procedure had to recognise the start and end of a command frame in a stream, as well as verify the checksum. Initially the parsing procedure was event driven, i.e. upon the GUI receiving data, the parsing routine was called. The software was validated using a test computer running the RTLinux GUI. Commands were sent from the test computer over either a TCP/IP or serial connection, and the stage 7 software was able to successfully identify individual commands and display them on a new line, showing the relevant command name, shown in Figure 10.

Figure 10. Stage 7 Software

Though the appearance of the GUI radically changed in following iterations, the stage 7 software was preserved as the GUI test program to validate the data output of the GUI at each stage.

2. Integration of UAV command functionality (Stage 14)

The RTLinux GUI is only capable of sending commands to the UAV; separate programs are required to display telemetry data. Whilst one of the project aims was for the GUI to display telemetry data, the first step
was to include the command functionality. It was at this stage that the advantages of selecting VB.NET were most apparent. Unlike the majority of other software languages, VB.NET offers a graphical design view, where the developer can see the application’s appearance. This is profoundly useful for GUI development; buttons and text fields can be placed in the form, and their properties changed as required, there is no need for complex procedures in the code that perform this at runtime. Effectively, the developer can see their interface while they are designing it. The graphical design view expedited the process of incorporating the command tabs into the GUI, as shown in Figure 11, at which point it had virtually the same functionality as the RTLinux version.

The GUI uses a colour-based acknowledgement system. After a command button is pressed, it changes colour to purple, if the GUI does not receive the correct UAV response in time, the command is resent until this occurs, at which point the command button colour changes to green. The lower portion of the interface is used to monitor the data sent by each command. In combination with the test program (Figure 10), the data output monitor was used to validate the functionality of each command. Every single command was systematically sent from both the RTLinux GUI and the stage 14 GUI to the test program and each command frame was compared to confirm that the stage 14 software functioned correctly, see Appendix B for the test data. Whilst some command buttons send the same data every time, others do not. The Proportional–Integral–Derivative (PID) update buttons convert a floating point number into four bytes, which are sent in the parameter field of the command frame. As each of the PID cell and row update buttons were tested with a variety of numbers and decimals throughout the cells, the accuracy of the float-to-byte conversion was assumed.

3. **Improvement of the parsing routine**

As well as running the RTLinux GUI, the test computer also has a copy of the Eagle autopilot software, meaning that the test computer can simulate both the original GUI and the UAV. The latter was instrumental in testing the command functionality. The UAV only sends receipts for legitimate commands, ensuring errors are identified in command frames, furthermore, when the GUI receives the command receipt, the internal handling of this event can be tested. Unfortunately, the original copy of the Eagle autopilot software on the test computer did not send telemetry data. When the GUI was tested with a real Eagle helicopter, the GUI became unresponsive, as the data parsing routine could not handle the volume of telemetry information sent from the UAV.

The parsing procedure was changed from an event-driven routine to a continuous loop. Although an event-driven procedure prevents the process from running unnecessarily, it is possible for a data-received-event to trigger the parsing routine whilst it is already operating. This could cause incoming data to be processed out-of-sequence or not at all. Now when the GUI receives data, it stores it in a buffer. When the parsing loop restarts, the buffer is analysed, sorted into telemetry information and command receipts, then cleared, making way for new data. The continuous parsing loop only processes the data in the order it was received, it cannot process it out-of-sequence, and it will only lose data if the buffer is full.

In order to prevent the parsing loop from wasting computer resources, The GUI was programmed to check for new data every 6 milliseconds and sleep during downtime. On the development computer this resulted in the CPU usage reduced from roughly 40% to 1% during inactivity, and 25% when processing telemetry, no adverse effects were noticed.
In order to effectively test the GUI, the test computer’s autopilot software was corrected to include a pseudo-telemetry feed. Two slight variations between the test computer and the actual eagle autopilot existed. Every 50th frame was erroneous, producing a checksum error; secondly, the telemetry was sent approximately 130 times a second, rather than 50. These variations were useful for testing the robustness of the GUI software, which proved capable in this extreme test scenario.

4. **Numerical display of the UAV telemetry feed**

Sensor and telemetry data onboard the UAV is sent to the GUI many times each second in the form of a *telemetry frame*. This follows the same format as a command frame, however, instead of command parameters, sensor data is sent. The GUI detects a telemetry frame based on its command code. Since the UAV can only communicate in bytes, where a byte can only have a value between 0 and 255, most of the raw sensor data must be encoded as a *series* of bytes before it can be sent to the ground computer. Furthermore, to aid the conversion process, if the raw data has decimal places, it is either directly converted into an array of bytes, or the number is multiplied by a power of 10 to produce an integer, and then converted. Since the sensor readings vary in the number of significant figures, the byte-conversion process varies, additionally, byte representations can be either signed or unsigned. Some types of telemetry frames are multiplexed, for example, there are too many GPS readings to send in one frame, so the readings are divided into several groups and one group is sent per telemetry frame, along with a multiplex index to identify the group. Finally, the bytes representing telemetry appear in the frame without any breaks that would help indicate beginnings and endings of readings.

Evidently, the process of decoding telemetry frames was exceedingly complicated. In order for the GUI to correctly decode the correct bytes and store the result in the correct location, a reference library was created that provides that GUI with the location of each reading in each type of telemetry frame, even if multiplexed, and a routine for decoding it. The routines are referenced by the start byte’s position in the frame, and the type of frame. This allows the telemetry decoder to progress through the frame, using the current position to call the relevant decoding routine. There are currently over 200 telemetry readings, each with a separate decoding routine. The result is a highly efficient telemetry decoding process that has minimal code length.

With the required software procedures in place, tabs displaying different sensor readings were incorporated into the GUI, as shown in Figure 12.

![Figure 12. Incorporation of telemetry display tabs](image)

5. **Incorporation of virtual instruments (Stage 17)**

Originally, four virtual instruments were planned to be incorporated into the GUI; an artificial horizon (AH), servo dials, a horizontal UAV locator, and a horizontal situation indicator (HSI). Only the AH and the locator have been included in the GUI to date, due to interface space restrictions, and project delays from the unforeseen complexity of telemetry handling. It was decided that both the AH and the locator should be prominently displayed in the GUI. This decision was based on commercial UAV GUI examples, as well as NATO STANAG 4586, section 3.1 *core requirements*, which stipulates the importance of UAV monitoring capability for a control system. By prioritising the display of attitude and location, the ability to monitor the UAV’s status from the
ground computer is maximised. The graphical displays replaced the GUI output box in the lower section of the interface, as shown in Figure 13, and the connection settings were also relocated to reduce the information density.

Each virtual instrument is an individual object that is ‘placed’ in the GUI in the design view. As a result, the instruments are completely modular, they can be relocated, resized and only require relevant telemetry inputs to function.

The UAV location is based on the State position x and y readings, aviation convention dictates that x is north-south displacement, whilst y refers to east-west displacement. The UAV location can be zeroed to change the datum point for the displacement reference. Additionally, the locator automatically changes scale to ensure that the location indicator is always in view, and the map range is used most effectively.

The AH was used with permission from the original author (http://tom.pycke.be), it represents the State roll and pitch readings.

D. Other Improvements

1. Settings Tab

Editing GUI settings from within the software code was found to be a tedious process, furthermore, the end user will be unfamiliar the code, making the task even more complicated. Settings cannot be hard-coded, as the variation in ground computers could require changes to communication configuration. For these reasons, the decision was made early in the design process that any settings that require changing on an even occasional basis would be made available within the GUI. In accordance with the information density guidelines of ISO 9241-143, the settings and GUI status readings were consolidated in a tab, rather than on the main display bar.

The settings tab, shown in Figure 14, allows the user to alternate between TCP/IP and serial connections, as well as specify connection settings. Additionally, the frequency of the data parsing loop can be set, and the telemetry displays can be individually toggled on or off. The settings tab also displays the number of telemetry frames received each second, the total checksum errors, and the current refresh rates of the telemetry displays.

Figure 13. Artificial Horizon and Locator

Figure 14. GUI settings tab
2. Automatic Settings Configuration

In an effort to reduce the level of user configuration required, and to account for changing computer resources allocated to the GUI, the display settings are automatically configured periodically, based on code execution speed. The time taken for the data decoder loop to execute is compared against the allocated time. If the execution time is less than that allocated, the thread sleeps for the remainder of the time, and the display quality settings are incrementally increased. If the execution has taken too long, then display settings are reduced in proportion to the over-run. The result is that within seconds, the GUI is able to reach an equilibrium between display quality and performance for even the slowest computers. Furthermore, in the event that the resources allocated to the GUI are changed, the display settings quickly adapt.

IV. Creation of GUI Adaptation Instructions

The final project deliverable was the creation of a set of instructions for future GUI development, specifically a guide for the incorporation of further telemetry tabs to display readings from future sensors. An indirect requirement of this deliverable was an open, easily adaptable software architecture, which was also a requirement of STANAG 4586.

As a result of the flexibility within the software architecture, the addition of further telemetry display tabs is straightforward, and there is a standard procedure. For this reason, the instructions for the adaption of the GUI are based around a tutorial to incorporate the attitude display tab. The tutorial clearly demonstrates the steps of adding the attitude readings to the reference library, along with the creating references for each reading, and a blank tab for the telemetry to be displayed in. No further work will be required from the user, as the GUI has been designed to automatically fill the display tab with the given telemetry, as well as applying formatting. By confining the required modifications to the reference library, the user will not have to make multiple additions to various files. The process has been made flexible, so that the user can apply custom formatting to a telemetry tab, similar to Figure 12. This is quite straightforward; the user only has to place text boxes for each telemetry reading, add reference tags, and the software makes all the other necessary changes.

There is also scope within the instructions for the addition of other virtual instruments. Since the instruments are modular, the user can inspect existing instruments, which will demonstrate how to create new instruments.

V. Conclusions

The functionality of the original RTLinux GUI has been recreated for the Windows operating system. The new GUI also incorporates telemetry displays, configuration settings, and an optimised display for every computer. An instruction for future adaptability is currently underway.

VI. Recommendations

As suggested by the adaptation instructions section, not all telemetry information is displayed by the GUI, however with the use of the guide, it will be easy to incorporate further display tabs. Additionally, it was identified that if the GUI was to be run on a computer with a wireless network card, a direct connection through the wireless card would be beneficial in bypassing the need for an external router. This would require more investigation into TCP/IP connectivity through a wireless network card with VB.NET.

Acknowledgements

I would like to thank my supervisor, Dr. Matthew Garratt for his guidance throughout my project, and most importantly for providing the tools I needed to create and test my software, I could not have completed this project otherwise. I would, again, like to acknowledge the author of the artificial horizon software used in my project, Tom Pycke.

References


Kaberry, G (2006). *PDA Control of UAV*. Canberra: UNSW@ADFA.


