Institutionen för datavetenskap
Department of Computer and Information Science

Final thesis

Rapid Development of Realistic UAV Simulations

by

Jonatan Rugarn

LIU-IDA/LITH-EX-G--09/003--SE

2009-02-26
Final Thesis

Rapid Development of Realistic UAV Simulations

by

Jonatan Rugarn

LIU-IDA/LITH-EX-G--09/003--SE

2009-02-26

Supervisors: Peter Fritzson, Paul Holmstedt
Examiner: Peter Fritzson
### Instrument Control Sweden (ICS)

ICS is a software company that develops NATO STANAG 4586 compatible ground station software for control of unmanned systems such as unmanned aerial vehicles (UAVs). To perform testing and demonstration of the ground station software ICS needs a realistic UAV simulator that implements the STANAG 4586 protocol. This thesis studies what methods are best suited for the rapid development of such a simulator.

One goal with the project was to examine what existing flight simulator systems and flight dynamics models can be used to rapidly develop a UAV simulator. Another goal was to design and implement such a simulator. It is found that it’s possible to quickly develop a UAV simulator based on existing projects such as the flight simulator FlightGear, the simulation framework OpenEaagles and the flight dynamics model (FDM) JSBSim.

The design of the simulator is modular, object-oriented and features real-time design techniques. The main application is a simulation of a Vehicle Specific Module, which implements the STANAG 4586 protocol. Another module based on the OpenEaagles framework simulates the aircraft and its subsystems. A third module consists of the JSBSim FDM and simulates the flight dynamics and movements of the aircraft under the forces and moments affecting it.
Abstract

Instrument Control Sweden (ICS) is a software company that develops NATO STANAG 4586 compatible ground station software for control of unmanned systems such as unmanned aerial vehicles (UAVs). To perform testing and demonstration of the ground station software ICS needs a realistic UAV simulator that implements the STANAG 4586 protocol. This thesis studies what methods are best suited for the rapid development of such a simulator.

One goal with the project was to examine what existing flight simulator systems and flight dynamics models can be used to rapidly develop a UAV simulator. Another goal was to design and implement such a simulator. It is found that it’s possible to quickly develop a UAV simulator based on existing projects such as the flight simulator FlightGear, the simulation framework OpenEaagles and the flight dynamics model (FDM) JSBSim.

The design of the simulator is modular, object-oriented and features real-time design techniques. The main application is a simulation of a Vehicle Specific Module, which implements the STANAG 4586 protocol. Another module based on the OpenEaagles framework simulates the aircraft and its subsystems. A third module consists of the JSBSim FDM and simulates the flight dynamics and movements of the aircraft under the forces and moments affecting it.

Sammanfattning

Instrument Control Sweden (ICS) är ett mjukvaruföretag som utvecklar NATO STANAG 4586 kompatibel markstationsmjukvara för styrning och kontroll av obemannade system som till exempel obemannade flygfarkoster (UAV). För att utföra testning och demonstration av markstationsmjukvaran behöver ICS en realistisk UAV simulator som implementerar STANAG 4586 protokollet. Det här examensarbetet studerar vilka metoder som är bäst lämpade för att snabbt utveckla en sådan simulator.

Ett av målen med projektet var att undersöka vilka existerande flygsimulatörer och flygdynamikmodeller som kan användas för att snabbt utveckla en UAV simulator. Ett annat mål var att designa och implementera en sådan simulator. Det befinns vara möjligt att snabbt utveckla en UAV simulator baserad på existerande projekt som flyg simulatören FlightGear, simuleringsramverket OpenEaagles och flygdynamikmodellen JSBSim.

Simulatorns design är modulärt uppbyggd, objekt orienterad och inkluderar realtidsdesignteknik. Simulatorns huvudapplikation är en simulering av en Vehicle Specific Module som implementerar protokollet STANAG 4586. En annan modul baserad på ramverket OpenEaagles simulerar flygfarkosten och dess underliggande system. En tredje modul, bestående av JSBSim, simulerar flygfarkostens flygdynamik och rörelser under de krafter och moment som påverkar den.
Acknowledgments

I would like to give thanks to Instrument Control Sweden for providing this exciting thesis work and to everyone working there for help and support with the project. A special thanks to Paul Holmstedt who has been my supervisor and to Henrik Wolkesson for much support with the STANAG 4586 SDK. Thanks to Per Månsson for technical support, to Henrik Salomonsson for proof reading this report, and to Jonas Yxfeldt for spreading joy and positive energy around the office. Also, I want to thank ICS for many fun moments such as Christmas parties and Unreal Tournament competitions. Finally, a thanks is in order to Paul and to Henrik Salomonsson for good badminton practice. I’m sorry I won every time.
Contents

Introduction ........................................................................................................................................ 12
Background ....................................................................................................................................... 12
Purpose ............................................................................................................................................ 12
Delimitations .................................................................................................................................... 12
Questions at Issue ............................................................................................................................. 12
Theoretical Frame of Reference ......................................................................................................... 14
Unmanned Aerial Vehicles ................................................................................................................ 14
Core UAV Control System ................................................................................................................ 15
Vehicle Specific Module ..................................................................................................................... 15
Data Link Interface ............................................................................................................................ 16
STANAG 4586 .................................................................................................................................... 16
Field Configuration ............................................................................................................................ 17
Vehicle Operating Modes ................................................................................................................... 17
Mission Commands ............................................................................................................................ 17
Vehicle Steering Command ................................................................................................................ 17
Loiter ................................................................................................................................................. 18
Flight Dynamics Model ..................................................................................................................... 18
Autopilot ........................................................................................................................................... 19
Method .............................................................................................................................................. 22
Exploring Existing Projects ................................................................................................................ 24
FlightGear .......................................................................................................................................... 24
JSBSim .............................................................................................................................................. 24
YASim ................................................................................................................................................ 25
LaRCsim .......................................................................................................................................... 26
UIUC FDM ........................................................................................................................................ 26
ODE .................................................................................................................................................. 26
OpenEaagles ...................................................................................................................................... 26
Design ................................................................................................................................................ 28
Aircraft .............................................................................................................................................. 28
VSM ................................................................................................................................................... 30
Implementation of the Air Vehicle ....................................................................................................... 32
Initialisation ....................................................................................................................................... 32
Flight Director Mode .......................................................................................................................... 33
Waypoint Mode ................................................................................................................................. 33
Waypoint Actions ............................................................................................................................... 33
Loiter Mode ....................................................................................................................................... 33
Simulating Errors ............................................................................................................................... 35
Connection between AV and VSM ...................................................................................................... 36
Implementation of the VSM ................................................................................................................ 38
Initialising the Simulation ................................................................................................................... 38
CUCS Authorization ............................................................................................................................ 38
Field Configuration Messages .......................................................................................................... 38
Mission Messages .............................................................................................................................. 39
Exiting the simulation ........................................................................................................................ 40
Results ............................................................................................................................................... 42
Test Results ....................................................................................................................................... 42
Test Display ....................................................................................................................................... 44
Concluding Remarks .......................................................... 45
Questions at Issue Answered ............................................... 47
Future Work ........................................................................... 47
Appendix A: Sources ........................................................... 49
Appendix B: Abbreviations .................................................... 51

Figures and Tables

Figure 1: The Sperver UAV ...................................................... 14
Figure 2: Overview of a UAV control system. .......................... 16
Figure 3: Loiter types ............................................................ 18
Figure 4: Table of comparisons between existing projects .......... 27
Figure 5: Basic UAV simulation dataflow overview. ................. 28
Figure 6: Overview of an OpenEaagles application .................. 29
Figure 7: Simplified UML diagram of the simulator application ... 31
Figure 8: Loiter pattern Figure 8 code. ................................... 35
Figure 9: Engine error code .................................................... 36
Figure 10: Field Configuration Response code ....................... 39
Figure 11: Commanded waypoint code .................................. 40
Figure 12: Using the console control handler ......................... 41
Figure 13: Loiter Figure 8 test ............................................... 43
Figure 14: Three options for implementing flight dynamics ....... 46
Introduction

**Background**

Instrument Control Sweden (ICS) is a small company located in Linköping. ICS develops ground station software for the control of unmanned systems. To perform testing and demonstration of the ground station software ICS wants to use a program that simulates an unmanned aerial vehicle (UAV).

**Purpose**

The purpose of the project is to examine what methods are best suited for rapidly developing realistic UAV simulations, and to develop a UAV simulator program. The simulator program needs to simulate a UAV as realistic as possible. This means that the program must simulate realistic flight dynamics. Common UAV functionality also needs to be implemented. This includes functionality to control the aircraft both by manual steering (Flight Director mode) as well as in autonomous modes (Waypoint and Loiter modes). The program needs to be able to communicate with ground stations in the same way that a real UAV would. To accomplish this the program needs to simulate a Vehicle Specific Module and implement the STANAG 4586 protocol.

**Delimitations**

It is important for ICS that some parts of the simulator project must not be open source. Therefore existing code that is licensed such that it requires the whole project to be open source must not be used. This is because ICS’ commercial STANAG 4586 SDK is used in the development of the simulator software. To ensure compatibility with the framework, ICS wants part of the project to be developed in C# with Visual Studio as development environment.

The simulator software does not need to provide functionality for landing the simulated aircraft, or taking off from the ground.

The time frame for the project is two and half months. Because of this, an already existing flight dynamics model (FDM) must be used.

**Questions at Issue**

The following questions need to be answered:

1. What major components are needed to rapidly develop a realistic UAV simulation system?
2. What method is suitable for rapidly developing a realistic UAV simulator?
3. What is a suitable design for a UAV simulator?
4. What already existing code or projects can be included to advantage when developing UAV simulations?
5. Which existing FDM is best suited for the rapid development of a realistic flight simulation?
Theoretical Frame of Reference

Unmanned Aerial Vehicles

Unmanned aerial vehicle (UAV) is a collective name for many kinds of unmanned aircraft. Most notably this includes fixed wing, and rotary wing air vehicles. These aircraft are typically controlled from a ground station through a radio link. UAVs are used for military, law enforcement and civilian purposes. Militarily, UAVs are mostly used for reconnaissance. An example of a civilian use of UAVs is for fire control.

An obvious advantage of using an unmanned aircraft instead of a manned aircraft is the safety of the persons operating the craft. Unmanned aircraft is also arguably cheaper to procure, compared to manned aircraft (Bone, 2003). The downside is that, according to Bone (2003) the accident rate of unmanned aircraft is 100 times higher than that of manned aircraft. This accident rate from 2003 is most likely lower today, but still much higher than that of manned aircraft. The advantages of unmanned aerial vehicles in certain situations, such as reconnaissance, are so big that they are used extensively.

An example of a UAV is the French Sperver, which is currently being used by the Swedish Armed Forces. The Sperver has a wingspan of 4.2 metres, a length of 3.3 metres and a height of 1.1 metres. It weighs 250 kg. The Sperver is equipped with sensors that provide the operator with real time video. (UAV Ugglan, 2007)

The Sperver is a medium sized UAV and it needs to be launched from a ramp (UAV Ugglan, 2007). Many UAVs are much smaller and some can be launched by hand, by throwing them much like a paper plane (Bone, 2003).

Figure 1: The Sperver UAV (UAV Ugglan, 2007).
Core UAV Control System

A Core UAV Control System (CUCS) is the main item in the architecture of a UAV Control System. A CUCS consists of hardware and software that, as a minimum, implements a data-link interface, the necessary Computer Software Configuration Items and CUCS/Human Computer Interface (STANAG 4586 Implementation Guideline Document, 2007). It provides a UAV operator with the functionality to control UAVs in all phases of a mission. It also provides a graphical interface to help the operator control the vehicle (Standard Interfaces of UAV Control System (UCS) for NATO Interoperability, 2007).

A CUCS controls and monitors aircraft by communicating with the vehicles’ Vehicle Specific Module (VSM). Each CUCS may be connected to several Vehicle Specific Modules, and each VSM may be connected to several CUCSes. To help coordinating the network of CUCSes and VSMs, Command, Control, Communications, Computer and Intelligence (C4I) nodes may be used. (STANAG 4586 Implementation Guideline Document, 2007)

A CUCS must be able to receive and send certain messages to/from VSMs. For example, a CUCS connect to a VSM by sending a CUCS Authorization request to a specific VSM, or broadcasting the request to every available VSM. It will then receive a response from the VSM indicating whether the VSM allows the CUCS to control or monitor it, and if so, on which Level of Interoperability (LOI). (STANAG 4586 Implementation Guideline Document, 2007)

After the CUCS has been authorized by the VSM, it may control and monitor the VSM’s vehicle and/or payloads at the allowed LOI. To control the air vehicle the CUCS needs to send other messages such as steering commands and mission upload commands, and receive messages, for example containing the vehicles inertial states or operating states. (STANAG 4586 Implementation Guideline Document, 2007)

Vehicle Specific Module

A VSM is a module that receives data from a Data Link Interface (DLI) and passes data to an air vehicle. It ensures compliance with protocols such as STANAG 4586, and acts as a bridge between the DLI and the air vehicle. A VSM can control one or more aircraft and a CUCS may connect to the VSM to take control of, or to monitor, an aircraft or a payload carried by an aircraft. (Standard Interfaces of UAV Control System (UCS) for NATO Interoperability, 2007)

A VSM may be located on the same machine as a CUCS, or on a different machine on the ground, or it may be split between a Ground Data Terminal, and an Air Data Terminal carried onboard an air vehicle. Each VSM may control several aircraft, as well as being connected to several CUCSes. When a VSM is connected to several CUCSes, only one of them can be controlling each aircraft at any given time. However, other CUCSes may receive data, and control payloads on the aircraft if the VSM allows it. (STANAG 4586 Implementation Guideline Document, 2007)

After the VSM has authorized a CUCS for a certain LOI, the VSM needs to be ready to receive other messages from the CUCS, such as configuration messages, and steering commands. The VSM also continuously sends status messages to authorized CUCSes, telling them about the air vehicles inertial states, operating states etc. (STANAG 4586 Implementation Guideline Document, 2007)
While a CUCS is a more general system, designed to be able to operate a large number of different craft, a VSM is, as the name implies, specific for a certain type of vehicle. (STANAG 4586 Implementation Guideline Document, 2007)

**Data Link Interface**

A DLI is the interface between CUCSes and VSMs. It provides for a standardized message set to enable communication between a wide variety of air vehicles and control stations (Standard Interfaces of UAV Control System (UCS) for NATO Interoperability, 2007). It is not specified what type of physical link the DLI must use. For example, the physical link might be a dedicated hardwire, or a satellite connection (STANAG 4586 Implementation Guideline Document, 2007).

![Figure 2: Overview of a UAV control system (STANAG 4586 Implementation Guideline Document, 2007).](image)

**STANAG 4586**

Standardization Agreement 4586 (STANAG 4586) is a NATO standardized protocol for the control of UAVs. Its purpose is to promote interoperability in the control and communication with UAVs for members of NATO and Partners for Peace (Standard Interfaces of UAV Control System (UCS) for NATO Interoperability, 2007). STANAG 4586 has become a de facto standard on the UAV market.

The protocol specifies a number of messages that may be transmitted between CUCSes and VSMs. These include messages for configuring, controlling and transferring data between CUCS, VSM, and the vehicles and payloads controlled by the VSM. Payloads may include for example sensors and weapons. The messages are typically transmitted on a Data Link Interface.
Each STANAG 4586 message has a message name, a numerical message id, and a number of fields containing various data. The messages are defined in a specification released by the NATO Standardization Agency (Standard Interfaces of UAV Control System (UCS) for NATO Interoperability, 2007). The STANAG contains about 80 different messages. Some of the more important ones are explained below.

Field Configuration
Since there are many different UAV systems in use that provide different functionality, the STANAG needs to provide a way for the CUCS to find out what functionality a certain air vehicle supports. This is done by the CUCS, after the initial authorization sequence, sending field configuration requests to the VSM. The VSM in turn responds with field configuration responses. These responses tell the CUCS what fields in what messages are supported by the VSM. The VSM can also set max and min values for fields, and high and low limit warnings and cautions. For enumerated fields, the VSM can specify vehicle specific enumerations that are not part of the STANAG. After the field configuration is complete the CUCS will typically alter its user interface so that only supported fields become accessible to the operator. (STANAG 4586 Implementation Guideline Document, 2007)

Vehicle Operating Modes
The STANAG 4586 protocol specifies a number of different vehicle operating modes for controlling an air vehicle (AV). The most common of these are: Flight Director, Waypoint, and Loiter. When flying in Flight Director mode, the operator steer the AV manually in close to real-time. When in Waypoint mode, the AV fly a mission defined by waypoints. In Loiter mode the AV is given a loiter point to which to fly. Upon arrival at the loiter point, the AV enters a loiter pattern around the point, while it waits for further instructions, such as a new loiter point. A CUCS sets an air vehicle’s operating mode through the STANAG message Vehicle Operating Mode Command. (Standard Interfaces of UAV Control System (UCS) for NATO Interoperability, 2007)

Mission Commands
When operating in Waypoint mode, the air vehicle navigates after a list of waypoints. A waypoint may have associated actions that the AV must perform upon arrival. Waypoints and waypoint actions are organized into missions. When a CUCS wants to upload a mission to an AV it sends all the waypoints, waypoint actions, and route messages defining the mission to the AV’s VSM. When the CUCS has finished transferring all the mission data it sends a Mission Upload/Download command, causing the VSM to load the mission into the AV’s computer. A CUCS can also request to download a mission from a VSM, in which case a similar sequence occurs except the direction of the messages are reversed. (Standard Interfaces of UAV Control System (CUCS) for NATO Interoperability, 2007)

Vehicle Steering Command
Steering of an air vehicle is performed through the STANAG message Vehicle Steering Command. This message allows the operator to set the commanded heading, altitude, speed, and waypoint number or loiter point. It also allows the operator to steer the vehicle in different ways through setting the commanded heading type. For example the operator may command a specific heading, or he may command a certain roll rate given in radians. (Standard Interfaces of UAV Control System (UCS) for NATO Interoperability, 2007)
Loiter

STANAG 4586 specify different loiter types that may be commanded by the CUCS if the air vehicle can implement them. These loiter types are shown in figure 3. It is possible to configure certain loiter parameters, such as loiter radius, bearing, loiter length etc through the Loiter Configuration message. (Standard Interfaces of UAV Control System (UCS) for NATO Interoperability, 2007)

![Loiter Types Diagram](image)

Figure 3: Loiter types (Standard Interfaces of UAV Control System (UCS) for NATO Interoperability, 2007).

Flight Dynamics Model

A flight dynamics model is a math and physics model that defines the movement of a simulated aircraft. A realistic FDM involves advanced math and physics. Naturally the complexity of an FDM program is also higher if the model is to be generic and usable to simulate the movements of many different types of aircraft.

The most common way of modelling flight dynamics is by building up the forces and moments that impact all of the aircraft’s axes. Some of these forces are environmental and may include gravity, atmospheric data, weather and air pressure. Other forces affecting the aircraft may come from engines, propulsion and aircraft controls. To realistically simulate how these forces and moments affect the movement of an aircraft, data about the aircraft’s metrics and other properties is needed.

There are many issues to be considered when incorporating a flight dynamics model into a simulation. Some of these issues are how to represent aircraft data, which coordinate system
and orientation model to use, and the level of complexity of the flight dynamics model. (Cooke, 1994)

Typically a set of motion equations calculates the aircraft’s movement. The vehicle state can be expressed in many different coordinate systems. Usually either body coordinates or earth coordinates are used. Body coordinates are generally based in the centre of gravity and follow the aircraft (Cooke, 1994). A common way to define a body coordinate system is such that the x-axis is aligned with a fuselage line along the aircraft, pointing forward from the craft’s nose. The z-axis points downward in a plane of symmetry, and the y-axis points to the side of the aircraft in a perpendicular plane such that a right-handed coordinate system is formed (Kroo, 2002). Earth coordinates are aligned with the earth and have their origin placed at a suitable location in the simulated world (Cooke, 1994). Generating the aircraft’s linear and angular velocities under the influence of the forces and moments are more easily performed in the body coordinates frame (Kroo, 2002).

The aircraft’s orientation can be expressed either in Euler angles, or with quaternions. Euler angles express the aircraft’s roll, pitch and yaw angles, and describe three successive rotations which brings the earth coordinates into alignment with the body coordinates. Quaternions are based on a unit sphere and can be described as a scalar plus 3-vector. Whether Euler angles or quaternions are best suited to express an aircraft’s orientation has been the source of heated debate. (Cooke, 1994)

When updating the aircraft’s position and orientation, this is usually done within an earth coordinates frame. This means that the aircraft’s linear velocities need to be transformed to earth coordinates. Rotating the velocity vector by the aircraft’s Euler angles performs the transformations. Likewise, updating the aircraft’s orientation is also done within an earth coordinate frame. Either the Euler angles or quaternions are used for these transformations. Using quaternions provide an elegant way of calculating the rotations through the use of four parameters. Three of the parameters describe the rotation axis and the forth parameter describe the angle through which to perform the rotation. (Cooke, 1994)

Naturally, a high level of realism is one goal when simulating flight dynamics. On the other hand a complete model including fully articulated control surfaces and airflow divergence patterns over the aircraft could seriously impact a simulation’s real-time capability. Therefore, the realism of a flight dynamics model must be weighed against its complexity. (Cooke, 1994)

If the flight dynamics model is specific for one type of vehicle, the aircraft data can be hard coded into the simulation. If a generic FDM is preferable, one needs to consider how to present the model with aircraft data. In most cases the program parses this data from input files. Developing a generic FDM is of course a much larger project since the model must be capable of simulating flight dynamics for aircrafts with a wide variety of different properties.

**Autopilot**

An autopilot (AP) is used to fly an aircraft autonomously. It may hold the craft on a certain course, altitude and speed, or it may fly a certain pattern, or navigate a list of waypoints. To accomplish this the autopilot needs to detect how the aircraft is moving, and adjust the ailerons, elevators and engines to set the aircraft on the desired heading, altitude and speed.
Typically a Proportional, Integral, Derivative (PID) controller is used to implement the autopilot functionality. The PID controller works by calculating a value to bring the aircraft from the current process value to the reference point. The explanation below of how a PID controller works is based on Control Theory 101 by Curtis L. Olson (2004).

The proportional device calculates this value based on the proportional difference between the process value and the reference point. A proportional controller will usually stabilize the aircraft well, but not at the exact value commanded. The reason for this is that as the vehicle is closing in on the reference point, the proportional error reaches a point where the adjustments made by the autopilot are very small.

The integral device calculates the integral of the error curve and adjusts the process value accordingly. This means that the longer the error exists, the more the AP will adjust to reach the reference point, thus effectively neutralizing the problem from the proportional device. Another problem that is introduced by the integral device is that the craft may oscillate as the AP keeps increasing the adjustment until it reaches the reference point. This causes the craft to continue past the reference point, and then needing to adjust back the other way.

The derivative device calculates the derivate of the error curve. This gives us the change rate of the error. Thus when the craft is closing in on the reference point, the derivative device will adjust the output value to reach the reference point without oscillating around it.

When the proportional, integral and derivative devices are combined they will become a smooth controlling mechanism. It takes some tuning to give each of the three devices the correct proportion compared to the others.
Method

During the initial phase of the project, various existing flight simulation projects were examined to find out whether they could be used in the development of the UAV simulator. This was done by searching for flight dynamics models, flight simulators, and simulation frameworks on the Internet. Reading about the various projects found made for a first selection of which ones might be interesting. The criteria for interesting projects were:

- Realism
- Performance
- Scalability
- Possibility to include in a distributed system
- User friendliness
- Which license the project is released under

Interesting projects was then downloaded and tested. Two different structures for the UAV simulator were considered:

- Integrating an FDM directly into the UAV simulator.
- Having a modular design which separates the VSM from the aircraft and flight simulation. The flight simulation could then be based on an existing flight simulation project.

Developing an FDM from scratch was not considered because of the complexity and time required for doing so. The modular structure was chosen because of its possibility to make use of existing flight simulation projects. Based on the criteria mentioned above, it was decided to integrate the OpenEagles framework and JSBSim into the simulator to provide a solid, realistic flight simulation. Utilizing these projects would also decrease the time required to develop the simulator.

The overall design was then drawn up based on three modules: VSM simulator, flight simulator, and flight dynamics model. An object-oriented design that provides several layers of fidelity was chosen to make the simulator robust, scalable and easy to understand. The VSM was designed to be the main application while the flight simulation would be a dynamically loaded library and the FDM a static library loaded by the flight simulator. The simulation of the VSM and the STANAG 4586 implementation would be based on ICS’ STANAG 4586 framework. This made for a realistic design where a CUCS can connect to, and communicate with, the VSM, and the VSM then communicates with the aircraft.

To save the time required to research aircraft data and implement a UAV input file, it was decided to originally use already existing aircraft, engine, and autopilot models that come with JSBSim. The JSBSim Cessna 172 model was chosen for this. If time allowed, an accurate UAV model could be implemented later.

A couple of different development methods such as the waterfall method, the spiral method and prototyping were considered for developing the simulator. Predictable methods like the waterfall method require a good understanding of all systems before implementation is started. It was decided that a rigid process method would be too time consuming, and not
necessary in a smaller project like this. Designing the whole project before starting implementation would require much reading in on the different projects and frameworks being used. Design flaws would most likely also be introduced that would then have to be corrected later. Instead a less formal bottom-up approach of evolutionary prototyping was chosen.

Prototyping means that a certain piece of the application is first developed, tested and refined. Then another piece is prototyped and added to the application, and so forth, making the application gradually evolve during continuous testing and implementation (Bell, 2005).

The basic components of the flight simulation were prototyped first and tested through keyboard and joystick input. After the basic components of the VSM was implemented, such as being able to communicate with a CUCS, testing was done by connecting to the simulator through the ground station software VISA and SkyView. Various STANAG 4586 messages were then sent from the ground station to the simulator. The simulators response was monitored both by its output and by setting breakpoints and examining received and sent data. A test panel for VISA was also developed specifically for testing the simulator.

Other ICS staff used the simulator to test ground station functionality being developed in other ICS projects. Indirectly, this also helped testing the simulator.
Exploring Existing Projects

During the initial phase of the project various existing projects was examined to decide if it would be advantageous to include any of them in the simulator software. It was especially important to find a realistic flight dynamics model to simulate the movements of the aircraft. During this phase, the Internet was used to search for projects of interest. A couple of interesting FDM projects was found: JSBSim, YASim, UIUC FDM and LaRCsim. The physics engine ODE was also of interest, as well as the flight simulator FlightGear and the simulation framework OpenEaagles.

**FlightGear**

FlightGear is an open source flight simulator. It comes with a great number of aircraft models, maps, and world data. It also implements an autopilot, provides a graphical interface, and uses a couple of different flight dynamics models to simulate the movements of the aircrafts. The flight dynamics models used by FlightGear are JSBSim (the default FDM), YASim and the UIUC model. LaRCsim was used as the default FDM in the early stages of FlightGear development, but this FDM was abandoned for the much more flexible JSBSim.

The FlightGear design is modular, and the modules can be independently compiled and then linked together into the FlightGear executable. Some of the independent modules include graphics, keyboard, joystick and other devices, and flight dynamics models. This modular design makes it rather easy to understand the code and also integrate new modules into it. (Sehgal, 2002)

An installation of FlightGear requires approximately 360 MB of hard disk space. The graphical interface is written in OpenGL, and the program requires a reasonable 3D accelerated graphics card to run smoothly. An Intel processor in the 2-3 Ghz range, or equivalent, is also required.

With the various FDMs, autopilot functionality, aircraft and world data, FlightGear is a highly realistic flight simulator. It reads input from configuration files and it can quite easily be included in larger projects by connecting to it through socket. This makes FlightGear rather flexible. However, much of the functionality is not always needed, and it might be a bit of overkill to include the full FlightGear program in building a UAV simulator. The program can be modified to run without certain modules such as graphics. FlightGear is licensed under the GPL.

After examining FlightGear, it was considered a very good option to use as a base for building a UAV simulation. It is stable, flexible, and highly realistic, provides autopilot functionality and is easy to integrate with another project through socket calls. However, since it is released under the GPL it would require the entire project to be open source. Therefore it was decided not to include FlightGear in this project.

**JSBSim**

JSBSim is a lightweight, data-driven, non-linear, six-degrees-of-freedom FDM application. Development of JSBSim started in 1996 by Jon S Berndt. The idea was to provide a flexible FDM that didn’t need aircraft data to be hard-coded, and instead could be configured through input files. (Berndt, 2008)
JSBSim simulates the flight dynamics by building up any number of specified coefficients from forces and moments that affect any of the axes of an aircraft (JSBSim Website, 2008). This makes for a highly realistic FDM in what might be called ‘normal’ flight situations (Basler, 2008).

Input data are presented to JSBSim through xml files describing aircrafts, engines and autopilots. Writing such aircraft models require knowledge in aerodynamics, flight mechanics and control theory. However, a number of xml aircraft models are available for download together with the source code.

JSBSim is open source and is written in C++. It can run as a standalone batch application or be included in larger project, in which case it may be compiled into a library file and called from the main application. Documentation is available through the JSBSim Reference Manual. The manual includes both a user’s manual and a programmer’s manual. This makes it relatively easy to write an interface to include JSBSim in a larger project, although it still requires some time to learn about how to do so.

The compiled JSBSim library takes approximately 15 MB of space on the hard drive, and run without any notable processor load.

JSBSim is released under the Lesser GNU Public License, and is used in a number of commercial and open source projects. The most notable of these are the open source flight simulator FlightGear and the framework OpenEaagles. Because JSBSim is released under the LGPL, it is possible to include JSBSim in a project, without releasing the other parts of the project as open source.

Because JSBSim is realistic, flexible, has good documentation and is released under the LGPL, it was decided that JSBSim would be used as the FDM for the UAV simulator. It is also the only open source standalone FDM project available.

**YASim**

YASim was written by Andrew Ross as an alternative FDM for FlightGear. Instead of using the coefficient buildup method, which is a more common way of implementing an FDM, YASim simulates flight dynamics based on geometric information. This makes it less exact, but ensures reasonably realistic behaviour in every situation. This might be compared with the coefficient buildup method, which might show strange behaviour in extreme or not flight-tested situations. (Basler, 2008)

The FDM was written specifically for the FlightGear flight simulator. The source code is available together with the full source for FlightGear. FlightGear’s source actually comes with Visual Studio solutions and project files, and YASim has its own project file. This makes it easy to quickly get an overview of the YASim code in the Visual Studio environment. However, one of the downsides with YASim is that there is no documentation available. Because of this, it is rather time consuming to become familiar with the code. Since YASim is part of FlightGear, the code is released under the GNU Public License.

Because of the lack of documentation, and because the GPL forces the whole project to be released as open source it was decided not to use YASim with the UAV simulator.
**LaRCsim**
LaRCsim is an FDM developed by NASA. It models six-degrees-of-freedom (6DoF) motions of the aircraft through the coefficient buildup method, and uses quaternions for the coordinate transformations. LaRCsim is less flexible than JSBSim and YASim, as it requires aircraft data to be hard-coded rather than reading it from a file. Source code for LaRCsim is available together with FlightGear. (Sehgal, 2002)

**UIUC FDM**
The Applied Aerodynamics Group at the University of Illinois at Urbana-Champaign originally developed an FDM to be used with FlightGear for the Smart Icing System Project. The aim of this project was to research how aircrafts are affected by flight in very cold weather (Sehgal, 2002).

UIUC FDM is based on LaRCsim, and uses the same methods to simulate the flight dynamics. The biggest differences are that the UIUC model is generic and reconfigurable. It loads its data from input files, and allows for new coefficients to be added into the equations. (Sehgal, 2002)

Some documentation for the UIUC FDM is available on the Smart Icing System website, and from papers written by Bipin Sehgal, Robert W. Deters and others. As with YASim the source code is only available for download with FlightGear, and because of the GPL license, it was decided not to use the UIUC model with the UAV simulator.

**ODE**
ODE is “a free, industrial quality library for simulating articulated rigid body dynamics” (Smith, 2006. pp 1). ODE is written in C and also has a C++ interface. It has built-in collision detection and is designed to be used in real-time simulations of moving objects (Smith, 2006). There is extensive documentation available on the ODE website.

ODE does not provide aerodynamics functions, but could be used as a basis for developing an FDM. Developing a flight dynamics model was estimated to take too much time, even with the help of ODE, and it was therefore decided that ODE is not suitable for the rapid development of realistic UAV simulations.

**OpenEaagles**
OpenEaagles is a simulation framework based on the Eaagles framework developed by the U.S. Air Force. Its purpose it to help “rapidly prototype and build robust, scalable, virtual, constructive, stand-alone, and distributed simulation applications” (OpenEaagles Website, 2009).

OpenEaagles is written in C++ and comes with ready Visual Studio project and solution files. The framework consists of a large number of classes for building applications, including aircraft classes, navigation systems, routes, waypoints, as well as math and unit conversion classes. It has a parser for reading input files describing objects to be created, such as aircrafts and their subsystems. The framework is designed to provide real-time functionality. The framework also has an interface to JSBSim for simulating flight dynamics. It features functionality to easily send data on a network and including a visualizer system. This makes it easy to build distributable and scalable applications with OpenEaagles.
There is extensible documentation available from the OpenEaagles website, including a number of white-papers, instruction videos, tutorials and example applications to quickly get started using the framework.

One of the downsides with OpenEaagles is that the third-party libraries included in the latest release are compiled with Visual C++ 2005 Express Edition, Service Pack 1, and are not compatible with a simulation application compiled with any other compiler. OpenEaagles is also not compatible with the latest JSBSim release. The OpenEaagles team has promised to also include third-party libraries compiled with Visual Studio 2008 in the next release, and an update tailoring OpenEaagles for the latest JSBSim release when JSBSim v1.0 is released. OpenEaagles is released under the LGPL license.

Because OpenEaagles is a real-time simulation framework, easily integrates with JSBSim and is well documented, it was estimated that using OpenEaagles would save much development time and also provide a solid base to build the aircraft simulation on. Because OpenEaagles is also released under the LGPL it was decided to use OpenEaagles to develop the flight simulation.

<table>
<thead>
<tr>
<th>Model / Project</th>
<th>Realism</th>
<th>Real-time</th>
<th>Generic</th>
<th>Autopilot</th>
<th>Hardware requirements</th>
<th>Documentation</th>
<th>Easy to integrate</th>
<th>License</th>
</tr>
</thead>
<tbody>
<tr>
<td>YASim</td>
<td>Medium</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Low</td>
<td>Not available</td>
<td>No</td>
<td>GPL</td>
</tr>
<tr>
<td>LaRCsim</td>
<td>High</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Low</td>
<td>Little</td>
<td>No</td>
<td>GPL</td>
</tr>
<tr>
<td>UIUC FDM</td>
<td>High</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Low</td>
<td>Some</td>
<td>Medium</td>
<td>GPL</td>
</tr>
<tr>
<td>JSBSim</td>
<td>High</td>
<td>Yes</td>
<td>Yes</td>
<td>Some functionality. Can be expanded.</td>
<td>Low</td>
<td>Very good</td>
<td>Medium</td>
<td>LGPL</td>
</tr>
<tr>
<td>FlightGear</td>
<td>High</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Medium</td>
<td>Some</td>
<td>Yes</td>
<td>GPL</td>
</tr>
<tr>
<td>OpenEaagles</td>
<td>High</td>
<td>Yes</td>
<td>Dependant on FDM</td>
<td>Dependant on FDM</td>
<td>Low</td>
<td>Very good</td>
<td>Yes</td>
<td>LGPL</td>
</tr>
<tr>
<td>ODE</td>
<td>High</td>
<td>Yes</td>
<td>Not an FDM</td>
<td>No</td>
<td>Low</td>
<td>Very good</td>
<td>Yes</td>
<td>GPL</td>
</tr>
</tbody>
</table>

*Figure 4: Table of comparisons between existing projects.*
Design

A number of things need to be taken into account when designing a simulation system. For example, does the system need to respond in real time? What performance criteria such as processor load are required? Does the system need to be scalable? Does it need to be distributed? Some of these factors might be conflicting with each other, and it needs to be decided which one is more important. If the level of realism is very high, the software might require a heavy processor load, or high memory allocation. If the simulation responds in real time, it may affect the level of realism etc. Usually there needs to be a compromise between these qualities.

To meet human in-the-loop latency deadlines, the application needs to respond close to real-time. Ironically, the closer to real-time the simulation responds, the more difficult it is to simulate realistic flight data. Obviously this is because realistic simulation means more complicated calculations, which in turn takes a longer time to perform. The OpenEaagles framework, which the flight simulator design is based on, provides abstract representation of system components, which makes it possible for the developer to intermix multiple levels of fidelity and tune the application to still run smoothly and meet human interaction requirements (Hodson, 2006).

For this project, a realistic simulation of both an air vehicle as well as the vehicle’s VSM is needed. The design of the UAV simulator is modular and separates the VSM, the flight simulation and the FDM. The flight simulation is developed in C++ with the OpenEaagles framework and compiles to a dynamically linked library (dll). The VSM is the main application. It communicates with ground stations and controls the aircraft through calling functions in the flight simulation dll and the flight simulator loads the FDM as a static library. The VSM is developed in C# with the help of ICS’ STANAG 4586 SDK.

Figure 5: Basic UAV simulation dataflow overview.

Aircraft

When simulating the air vehicle, characteristics such as metrics, flight dynamics, and flight control system, engines, fuel system, and autopilot needs to be considered. It is possible to model all these properties with JSBSim and OpenEaagles.

For the UAV simulator, we want it to respond in real-time to provide the operator with a scenario as close to flying a real air vehicle as possible. This means that it must be ensured
during implementation of the simulated aircraft that time-critical tasks not get interrupted. A more low-level control in the program code is therefore desirable.

The OpenEaagles framework runs two or more threads. One of these threads handles all time-critical tasks, and the other threads handle all other tasks.

The main system object in an OpenEaagles application is the Station class, which handles input/output and manages the applications threads. The Station class has a Simulation object which manages simulation data such as participating players (aircraft etc). Figure 6 shows an overview of OpenEaagles’ architecture:

Figure 6: Overview of an OpenEaagles application (OpenEaagles Website, 2009).

As is shown in figure 3, a player contains objects such as autopilot model, navigation systems and flight dynamics model. The navigation system in turn has access to route and waypoint information. The AP model provides an interface for autopilot functions such as holding heading, steering to waypoints, or loiter. It doesn’t however control the craft directly, but calculates commanded values that are then passed to the FDM. The FDM must then actually implement the autopilot. This means that the program is dependent on using a JSBSim autopilot.

Information about participating systems, players, and other classes and data can be parsed by an OpenEaagles class from an input file. This makes it easy to alter or change the simulation runs without changing the program code or needing to recompile.
The flight simulation also features a number of dll exportable functions that are meant to be called by the VSM. These functions set various simulation and aircraft settings, and returns aircraft data to the VSM.

**VSM**

To realistically simulate a VSM the most important aspects are the implementation of STANAG 4586, the functionality to communicate with a CUCS, and the connection to the simulated air vehicle.

The basic strategy when designing the VSM was to implement as much functionality as possible in the VSM application, and have as little coupling as possible to the flight simulation. Only when it’s necessary should the VSM call functions in the flight simulation library. This separation of VSM and air vehicle makes the code more manageable and is also realistic because it simulates the interaction between a real VSM and air vehicle. Much of the functionality concerning implementation of the STANAG was also easier to develop in C# since a C# STANAG 4586 SDK is used to handle STANAG messages.

The main building block in the VSM design is the VSM class. This class handles events that occur upon reception of STANAG 4586 messages. The messages that only concern the VSM part of the simulation are handled directly by the message handlers. Messages containing data that concern the steering or operating of the vehicle generally calls a wrapper function in the FlightSimulator class.

The VSM class handles all communication with CUCSes, and keeps track of such connected and authorized CUCSes. The VSM class also keeps track of missions, routes, waypoints, and maintains a flight simulation object and a field configurations object. It updates information about the air vehicles states at a close interval, and sends such information to authorized CUCSes.

The FlightSimulator class provides the interface toward the flight simulation dll. A third class manages STANAG 4586 messages field configuration settings. A simplified UML diagram of the simulator application is shown in figure 7.
Figure 7: Simplified UML diagram of the simulator application. The VSM is the main executable. Its class SimulationManager provides an interface to the flight simulation library. The flight simulation in turn has an interface towards JSBSim.
Implementation of the Air Vehicle

During the implementation of the simulated aircraft a number of issues needed to be solved. Among these were:

- Steering the aircraft manually
- Flying routes after waypoints
- Performing actions at waypoints
- Loiter at given coordinates
- Simulate errors on the aircraft
- Communicating with the VSM

Some of this functionality was already implemented in existing OpenEaagles classes, while other functionality needed to be added. Subclasses were made from the OpenEaagles’ Station, Player, Autopilot and Action classes to extend the functionality in corresponding areas. The extended functionality especially concerned making the simulated air vehicle STANAG 4586 compatible. For example, functionality to configure loiter settings, and operating the vehicle in different operating modes needed to be added.

Initialisation

Calling its exec function starts the flight simulator. The exec function reads the OpenEaagles input file that describes the objects to be created. These objects include aircraft, navigation system and flight dynamics model etc. The input file also specifies the aircrafts initial latitude, longitude and altitude. The OpenEaagles parser loads this initial position data into JSBSim when it starts the FDM.

A conflict occurred because the main simulator application also has its own input file containing various options, including air vehicle initial position. This input file is the main simulation configuration file, and its content must override the contents of the OpenEaagles input file. Therefore the VSM, after starting the flight simulator needs to reposition the air vehicle at the overriding position.

Repositioning the air vehicle turned out to be a problem because OpenEaagles updates the air vehicles altitude from JSBSim during each frame, and it is only possible to set initial altitude in JSBSim (i.e. after initial position has been set, it’s not possible to reposition an air vehicle in JSBSim. OpenEaagles is dependent on JSBSim for altitude data, but not for latitude and longitude data).

OpenEaagles still provides functions for setting new altitude, as well as latitude and longitude. Finding out why and where the new altitude was overridden with the old one was rather tricky, since it was difficult to debug the different threads running simultaneously. The code causing the problem was found by closely examining the code in the Player class. Once the problem was located it could be solved by resetting initial position data in OpenEaagles, and then restarting JSBSim.

Solving this problem required reading up on JSBSim. It demonstrates that it is important to acquire some knowledge about JSBSim programming and development, even when using JSBSim through an existing interface such as OpenEaagles.
**Flight Director Mode**

During the implementation of manual steering mode Flight Director, the autopilot was at first turned off and steering was conducted by direct control of the stick. When steering by direct stick control, the aircraft was very unstable and easily drifted in various directions as soon as the control stick was released. This is highly realistic but makes manual steering without autopilot suitable only when keeping the plane under constant control through a joystick. It is desirable that an operator can set the plane on a certain course, and then release the joystick while the AV holds the commanded course.

The Flight Director mode was instead implemented by keeping the AP turned on, and allowing commanded heading and altitude to be set. This caused some other problems: STANAG 4586 allows to set a commanded turn rate, but the JSBSim autopilot used does not provide a function to set commanded roll. Therefore the turn rate field was not implemented. It might be possible to extend the autopilot with a commanded roll controller.

**Waypoint Mode**

The Waypoint mode was quite easily implemented in OpenEaagles by using the existing Navigation, Route and Autopilot classes. A player’s Navigation object has a route into which steerpoints can be inserted.

The Autopilot class have the functionality to steer the vehicle in different modes such as navigation mode, loiter mode, and follow-the-lead mode. When Navigation Mode is set in the autopilot, the air vehicle automatically navigates after the steerpoint data in the route object.

**Waypoint Actions**

The STANAG 4586 protocol allows to set actions to be performed at waypoints through messages AV Loiter Waypoint, Payload Action Waypoint, Airframe Action Waypoint and Vehicle Specific Waypoint. Currently the only action supported by the simulated air vehicle is Loiter Action.

The OpenEaagles framework provides a way of implementing waypoint actions by using the Action class. A Steerpoint (OpenEaagles notion for waypoint) has an Action pointer, and calls the Action’s trigger function upon arrival at its position. The Action class is an abstract baseclass that can be subclassed from to create various waypoint actions. A subclass was made from the Action class to create a LoiterAction class. The LoiterAction’s trigger function requests a loiter at the air vehicle’s current position.

**Loiter Mode**

Loiter requests can be made to an air vehicle when its OpenEaagles autopilot has been set to loiter mode. While STANAG 4586 allows loiter configuration for loiter direction, loiter bearing, loiter length, loiter radius and loiter type, it was only possible to set the loiter length and loiter direction in the OpenEaagles autopilot class. The loiter function only implemented a racetrack pattern.

The loiter functionality was overridden by the new class FlightSimulatorAutopilot. The new class allowed for all STANAG 4586 loiter configuration fields to be set, and added a new loiter type: Figure 8.
The figure 8 loiter pattern was implemented by calculating the length to the focus points from the anchor point. A fly-to point was then placed on each side of the focus point, on the distance of the loiter radius. The bearing from the loiter point to these fly-to points was calculated by taking the arctangent of the distance to the focus point and the loiter radius. This is demonstrated in figure 8.

definitions:

```cpp
double radius = 0.053833866 * speed * speed * 3; // turn radius (feet)
if(loiterRadius > radius)
    radius = loiterRadius;
double radiusNM = radius * Basic::Distance::FT2NM;

// Offset (width) of the orbit pattern (ft)
double orbitOffsetFt = 2.0 * radius;
double orbitOffsetNM = orbitOffsetFt * Basic::Distance::FT2NM;

// Pattern length
double orbitLengthFt = loiterLength * Basic::Distance::NM2FT;
if (orbitLengthFt < orbitOffsetFt)
    orbitLengthFt = orbitOffsetFt;

// Distance from anchor point to focus point (NM)
double distFP = orbitLengthFt / 2 * Basic::Distance::FT2NM;
double distFt = distFP * Basic::Distance::NM2FT;

// Distance from anchor point to the actual fly-to point (pythagoras of radius and distance to focus point).
double distLoiterPointNM = sqrt(distFP * distFP + radiusNM * radiusNM);

// Relative bearing. The angle between anchor, focus, and loiter point. Arctangent function.
double relBrgDeg = atan2(radius, -distFt) * Basic::Angle::R2DCC;

double trueBrgDeg = 0;
double lat2, lon2;

// Calculate waypoint pattern
// point 1
// Check if we want counter clockwise turn
if (ccwFlg) {
    trueBrgDeg = crs - relBrgDeg;
} else {
    trueBrgDeg = crs + relBrgDeg;
}
double bearingPointOne = trueBrgDeg;

Basic::Nav::gbd2ll(ala1t, alon, trueBrgDeg, distLoiterPointNM, &lat2, &lon2);
figureEightLat[0] = lat2;
figureEightLon[0] = lon2;

// point 2
if (ccwFlg) {
    trueBrgDeg = crs + relBrgDeg;
} else {
    trueBrgDeg = crs - relBrgDeg;
}
double bearingPointTwo = trueBrgDeg;
```
Basic::Nav::gbd2ll(alat, alon, trueBrgDeg, distLoiterPointNM, &lat2, &lon2);
figureEightLat[1] = lat2;
figureEightLon[1] = lon2;

//point 3
trueBrgDeg = bearingPointTwo - 180;
Basic::Nav::gbd2ll(alat, alon, trueBrgDeg, distLoiterPointNM, &lat2, &lon2);
figureEightLon[2] = lon2;
figureEightLat[2] = lat2;

//point 4
trueBrgDeg = bearingPointOne - 180;
Basic::Nav::gbd2ll(alat, alon, trueBrgDeg, distLoiterPointNM, &lat2, &lon2);
figureEightLon[3] = lon2;
figureEightLat[3] = lat2;

Figure 8: Loiter pattern Figure 8 code.

Simulating Errors
The UAV simulator can be run with the option to randomly simulate errors in the AV and VSM. The error rate can be set to allow errors to happen more or less frequently. The error simulation was implemented by having a function in the air vehicle randomly test for errors during each frame. If an error occurs, the function randomly picks a subsystem to generate an error for. An example of an error that might occur is an engine error, which causes the engine to either shut down or not run on full power. Example code below shows the implementation of the engine error.

```c++
void SimPlayer::generateError(int subsys)
{
    switch( subsys )
    {
        case 0: engineError(); break;
        case 1: mechanicalError(); break;
        case 2: electricalError(); break;
        case 3: commError(); break;
        case 4: propulsionError(); break;
        case 5: navigationError(); break;
        case 6: payloadError(); break;
        case 7: recoveryError(); break;
        case 8: environmentalCSErr0r(); break;
        case 9: VSMError(); break;
        case 10: ADTErr0r(); break;
        case 11: GDTErr0r(); break;
        default: break;
    }
}

void SimPlayer::engineError()
{
    std::cout << "*************************************\n"
    std::cout << "Engine ERROR\n"
    std::cout << "*************************************\n"
    int r = rand() % 4;
```
LCore throttle_positions[1];

if(r == 0)
{
    subsystemsStatus.Engine = Failed;
    throttle_positions[0] = -1.0;
    this->setThrottles(throttle_positions,1);
}
else if(r == 1)
{
    subsystemsStatus.Engine = Emergency;
    throttle_positions[0] = 0.0;
    this->setThrottles(throttle_positions,1);
}
else if(r == 2)
{
    subsystemsStatus.Engine = Warning;
    throttle_positions[0] = 1.0;
    this->setThrottles(throttle_positions,1);
}
else
{
    subsystemsStatus.Engine = No_Status;
    throttle_positions[0] = 2.0;
    this->setThrottles(throttle_positions, 1);
}

\textit{Figure 9: Engine error code.}

\textbf{Connection between AV and VSM}

The connection between the simulated air vehicle and the VSM is provided by a number of exportable functions in the flight simulator. The class FlightSimulatorManager on the VSM side provide the wrapper interface towards the flight simulation library. This setup means that the flight simulator never calls the VSM. The VSM needs to call the flight simulator every time it wants to send data to, or receive data from the flight simulator. The VSM therefore calls the flight simulator during each frame, to receive updates on the aircraft’s inertial states, position data, operating states etc.

Other examples of dll-exportable functions provided by the flight simulation library are functions for setting the vehicle operating mode, sending vehicle steering commands and configuring loiter parameters. These wrapper functions in turn calls public functions in the Station and Player classes, which implements the necessary functionality.
Implementation of the VSM

Upon connection from a CUCS, the VSM software sets up event handlers for reception of STANAG 4586 messages. Some of the STANAG messages are trivial to handle. As much handling as possible is taken care of without calling any functions in the flight simulator. I.e. separation and encapsulation of functionality between the VSM and flight simulator is emphasized.

Initialising the Simulation

When the simulator application starts up, it reads an input file containing various options and the simulated vehicles initial position. The SimulatorVSM constructor starts a new thread which calls the FlightSimulator’s exec function. This function in turn starts a time-critical thread, which performs all time-critical tasks. It then enters an infinite loop that updates all non time-critical data each frame.

CUCS Authorization

When a CUCS wants to connect to the VSM it must send a CUCS Authorization request. The VSM then checks which LOI the CUCS is requesting. If the CUCS is authorized and it is requesting control of the air vehicle, the VSM will grant control to the CUCS unless another CUCS is already controlling the AV. If the AV is under control from another CUCS, the new CUCS may request to override the other CUCS. This will be granted unless the other CUCS has also requested override. In the event that the new CUCS is not granted control of the air vehicle, it will instead be granted a LOI to monitor the aircraft. A CUCS may also request control of any payload controlled by VSM. This works in the same way as requesting control of the air vehicle itself.

To implement the CUCS authorization routine the VSM must keep track of which CUCS is currently controlling the air vehicle and whether it has requested override. Storing a reference to the controlling CUCS easily accomplish this.

Field Configuration Messages

Some of the more complicated STANAG 4586 messages to handle include the messages related to field configurations. The VSM needs to be able to respond to any field configuration request from a CUCS with the appropriate values. This requires a rather large quantity of code. To manage the field configurations, a class FieldConfiguration was created. This class has a list of all supported field elements, and their settings. Enumerated elements in turn have a list of all enumerations. The list of supported fields is initialised in function called by the VSM’s constructor.

When the VSM receives a field configuration request, it only needs to search in the field configuration class for the appropriate element, and can then retrieve all the correct settings for that field. An example of this is shown in figure 10.

```java
FieldConfigurationElement fieldConfigurationElement =
    fieldConfiguration.getFieldConfigurationElement(dataElementName);
if (dataElement is DoubleDataElementSettings || dataElement is
    FloatDataElementSettings)
{  
```
FieldConfigurationDoubleResponse doubleResponse = (FieldConfigurationDoubleResponse)cucs.CreatePreparedDliMessage(MessageTypes.FieldConfigurationDoubleResponse);
doubleResponse.FieldSupported = 0;
doubleResponse.RequestedMessage = message.RequestedMessage;
doubleResponse.RequestedField = message.RequestedField;

if (fieldConfigurationElement != null)
{
    doubleResponse.FieldSupported = 1;
    doubleResponse.MinValue = fieldConfigurationElement.MinValue;
    doubleResponse.MaxValue = fieldConfigurationElement.MaxValue;
    doubleResponse.HighWarning = fieldConfigurationElement.HighWarning;
    doubleResponse.LowWarning = fieldConfigurationElement.LowWarning;
    doubleResponse.HighCaution = fieldConfigurationElement.HighCaution;
    doubleResponse.LowCaution = fieldConfigurationElement.LowCaution;
}

response = (IMessage)doubleResponse;

Figure 10: Field Configuration Response code.

Mission Messages

The Waypoint Mode was implemented by letting the VSM handle all mission upload and download requests. The VSM keeps two mission objects. One contains the current mission, and one stores new incoming mission data (waypoints, routes and waypoint actions). On receiving a Mission Upload/Download message with the Mission Plan Mode set to Load, the VSM copies the incoming mission to the current mission and clear the incoming mission.

When a waypoint number is commanded by the CUCS, the VSM checks whether the commanded waypoint lies in the current route. If the commanded waypoint is not in current route, the VSM searches through the current mission to find all the waypoints in the same route as the commanded waypoint. The waypoints in the new commanded route are then uploaded to the simulated air vehicle in the correct order. This is demonstrated in the code below:

void OnVehicleSteeringCommand(IVehicleSteeringCommand message, BaseCUCSConnection cucs)
{
    try
    {
        VehicleSteering vehicleSteering = new VehicleSteering();
        if (currentControlMode == VehicleControlModes.Waypoint)
        {
            ushort commandedWaypoint = message.CommandedWaypointNumber;
            if (commandedWaypoint == 0)
                return;
IAVRoute commandedRoute =
currentMission.IsUsedInAnyRoute(commandedWaypoint);
if (commandedRoute == null)
    return;

if (commandedRoute != currentRoute)
{
    //Load commanded route to FlightSimulator
    IAVPositionWaypoint firstWaypoint =
        currentMission.GetIAVPositionWaypoint(
            commandedRoute.InitialWaypointNumber);
    flightSimulator.PositionWaypoint(firstWaypoint);
    ushort wpIdx = 1;

    IAVLoiterWaypoint loiterWaypoint =
        currentMission.GetIAVLoiterWaypoint(
            commandedRoute.InitialWaypointNumber);
    if (loiterWaypoint != null)
        flightSimulator.LoiterWaypoint(loiterWaypoint, wpIdx);

    IAVPositionWaypoint nextWaypoint =
        currentMission.GetIAVPositionWaypoint(
            firstWaypoint.NextWaypoint);
    while (nextWaypoint != null && nextWaypoint != firstWaypoint)
    {
        wpIdx++;
        flightSimulator.PositionWaypoint(nextWaypoint);
        if ((loiterWaypoint =
            currentMission.GetIAVLoiterWaypoint(
                nextWaypoint.WaypointNumber)) != null)
            flightSimulator.LoiterWaypoint(loiterWaypoint, wpIdx);

        nextWaypoint =
            currentMission.GetIAVPositionWaypoint(
                nextWaypoint.NextWaypoint);
    }

    //AddNewRoute tells flightsimulator to discard old route and load the new
    //list of waypoints
    flightSimulator.AddNewRoute();
    currentRoute = commandedRoute;
}

ushort indexInRoute = getWaypointIndexInRoute(
    currentRoute, commandedWaypoint);
vehicleSteering.CommandedWaypointNumber = indexInRoute;

flightSimulator.OnVehicleSteeringCommand(vehicleSteering);

Figure 11: Commanded waypoint code.

Exiting the simulation

The simulation runs as a console application. A problem occurred when trying to exit the
application cleanly. When typing Control-C, Windows shut down the application without any
of the applications exit code getting executed. This left running threads and memory leaks.
Using the win32 console control handler API solved the shutdown problem. A handler function was implemented to make windows call the simulator’s exit code on shutdown of the simulator console application.

```csharp
[DllImport("Kernel32")]
public static extern bool SetConsoleCtrlHandler(HandlerRoutine Handler,
bool Add);

// A delegate type to be used as the handler routine
// for SetConsoleCtrlHandler.
public delegate bool HandlerRoutine(CtrlTypes CtrlType);

// An enumerated type for the control messages
// sent to the handler routine.
public enum CtrlTypes
{
    CTRL_C_EVENT = 0,
    CTRL_BREAK_EVENT,
    CTRL_CLOSE_EVENT,
    CTRL_LOGOFF_EVENT = 5,
    CTRL_SHUTDOWN_EVENT
}

public static bool ConsoleCtrlCheck(CtrlTypes ctrlType)
{
    if (simulatorVsm != null)
    {
        simulatorVsm.Shutdown();
        Environment.Exit(0);
        return true;
    }
}

try
{
    //We add a console ctrl handler to be able to exit the console application
    //cleanly
    SetConsoleCtrlHandler(hr, true);
    Application.Run();
}
```

Figure 12: Using the console control handler.
Results

The purpose of this project was to examine methods for rapidly developing realistic UAV simulations, including what existing flight dynamics models and other related projects could be used to advantage during such development. Another goal was to develop a STANAG 4586 compatible UAV simulator.

The project resulted in an analysis of a number of flight dynamics models and other projects (Exploring Existing Projects). The projects examined are well-known open source projects. There may be other projects, both open source and commercial, that are suitable to use in the development of a UAV simulation.

For this project a commercial STANAG 4586 SDK was used during the implementation of the simulated VSM. This report does not discuss other solutions for implementing a VSM. Without the help of an existing framework, designing and implementing a VSM simulator can be a rather large project.

The goal of developing a UAV simulator was accomplished, although some improvements are needed (see chapter 9.2 Future Work). The simulator program simulates realistic flight dynamics and can simulate errors. CUCSes can connect to the simulator program and take control of the simulated aircraft through the STANAG 4586 protocol.

Test Results

Testing of the UAV simulator was done continuously during the implementation phase of the project. Towards the end of the project more structured testing was also performed. Great use was made of the Visual Studio debugging functions during all stages of testing.

During the initial development of the aircraft simulation, the FlightSimulator module was compiled to an executable to make it possible to debug. A visualizer called SimpleOTW was also connected to FlightSimulator to make testing easier by generating a graphical output view.

After the VSM was developed much of the testing was performed by utilizing ICS’ ground station software VISA and SkyView. Various STANAG messages were then sent from these programs to the simulator and the results were monitored on the ground station software GUI.

Test cases included:

- Loiter figure 8. The AV was expected to perform a correct figure 8 pattern when approaching the loiter point from any angle. Initially the AV only performed a correct figure 8 pattern when approaching the loiter point from certain angles. This revealed a flaw in how the figure 8 fly-to points were placed, which was consequently corrected and tested again successfully.
- Loiter racetrack. Performed in the same way as the test case for loitering a figure 8 pattern. This test was successful.
- Performing a waypoint mission. A waypoint mission was uploaded to the AV. The AV was then commanded to fly to one of the waypoints. The result expected was for the AV to successfully navigate to the commanded waypoint and, upon arrival, automatically start navigating towards the next waypoint in the mission. This test was successful.
• Loiter at a waypoint. A waypoint with an associated loiter action was uploaded to the AV. The AV was then commanded to fly to the waypoint. The expected result was for the AV to enter a loiter pattern upon arrival at the waypoint. This test was performed successfully.

• Commanding heading and altitude in Flight Director mode. The AV was expected to adjust its ailerons and elevators to reach the reference point. This test revealed some problems with the aircraft model and autopilot. When the AV was pitching upwards the velocity quickly dropped to a level where the craft started to lose altitude, which caused the AP to pitch up even more and so forth. Editing the aircraft’s engine data to make the engine somewhat stronger solved this problem.

Figure 13 illustrates how a typical test case was conducted. The figure 8 loiter pattern was tested by sending a vehicle steering command telling the AV to loiter at given coordinates. The response was then monitored in the VISA graphical user interface. As seen in the figure the AV is flying a figure 8 pattern, and this test case was considered successful.

![Loiter Figure 8 test](image)

Figure 13: Loiter Figure 8 test. The figure shows the graphical user interface in VISA.

Other functionality, such as CUCS authorization and uploading and downloading mission data, was tested by debugging incoming messages and examining the VSM’s response line by line.
Other ICS staff used the UAV simulator for testing of ground station software being
developed. This was very helpful since it helped discover bugs and clear out
misunderstandings and interpretation issues about the STANAG.

**Test Display**
Because ICS ground control station SkyView is still under development some CUCS
functionality needed for testing the simulator had yet not been developed. Therefore a special
test panel was developed. The test panel implements part of the functionality of a CUCS and
it can send STANAG 4586 messages to connected VSM’s. It compiles into a dll and may be
loaded by VISA and SkyView. The test panel can be seen in Figure 13. It features controlling
an AV in the operating modes Flight Director, Waypoint and Loiter, configuring loiter
parameters and uploading missions to an air vehicle.
Concluding Remarks

Developing a realistic UAV simulation requires a real-time flight simulation architecture to meet human interaction requirements. An object-oriented design is strongly preferable to make the simulator flexible, scalable and easy to understand. The simulator also requires a realistic flight dynamics model, an autopilot, and the implementation of a VSM.

There are a few methods to choose from when incorporating realistic flight dynamics in a UAV simulation. The first option would be to develop a flight dynamics model. If the simulator only needs to be capable of simulating a limited number of aircraft types, the aircraft data could be hard coded into the FDM. With good knowledge in mathematics, physics, aerodynamics and flight mechanics developing such a model would take roughly between a few weeks to a couple of months depending on how advanced the model is.

A second option would be to incorporate an existing FDM into the simulator. A couple of flight dynamics models are available with the open source flight simulator FlightGear. Unfortunately these lack good documentation. The time required to learn how to use these models and how to incorporate them into a larger project therefore increases.

JSBSim would be the best option for incorporating an existing flight dynamics model into a UAV simulation. JSBSim is stable and realistic in most situations. It is designed to either run in standalone batch mode, or to be integrated with a larger project. Integrating JSBSim is made easier by the extensive documentation available in the JSBSim Reference Manual.

The JSBSim FDM is dependant on accurate air vehicle data such as vehicle metrics, mass balance, propulsion and engines etc, to realistically simulate the vehicle’s flight dynamics. This makes JSBSim very realistic, but it also requires the developer to research this data and provide it in an xml file describing the aircraft. JSBSim comes with many existing aircraft and engine files. It is possible to edit and utilize one of these that resemble the craft to be simulated. Of course doing so makes the simulation less exact. Altogether using JSBSim saves much time in developing a realistic flight simulation. Since JSBSim is generic and reads its input from xml files, it also makes the simulator program flexible.

A problem when using JSBSim is that the existing autopilots do not work very well. The elevation controller is very slow and overshoots, and the wing leveller overcompensates. There is also no velocity controller in the autopilot. With the right knowledge it is quite easy to extend the autopilot, which is defined in an xml file. There is also documentation available on how to do so. Extending the autopilot requires knowledge in control theory and aerodynamics. It also requires a good understanding of how JSBSim works.

The third option for simulating realistic flight dynamics is to make use of an existing simulation framework or flight simulator together with a flight dynamics model. OpenEaagles and FlightGear are two such suitable options. The time required to develop a robust real-time simulation system is much shorter when using such a solution. This is partly because these projects already have classes in place representing many systems such as aircraft, navigation systems, waypoints and routes. Naturally not having to design and implement these classes saves much time. They also incorporate realistic flight dynamics models and are built on solid real-time technology, which would otherwise take much time to design, implement and test.
Figure 14: Three options for implementing flight dynamics. The FDM can be developed from scratch as a part of the simulator application. An existing flight dynamics model can be integrated with the simulator application as an external module. Also, an existing flight simulator can be integrated with the simulator application, and the flight simulator can integrate with an existing FDM project.

FlightGear is a mature flight simulator that also has a solid autopilot. Using FlightGear as the foundation for rapidly developing a UAV simulation would probably be the best choice, but for this project FlightGear was not an option because of licensing issues.

OpenEaagles provides a flexible framework to use as a base when developing simulations. During development of the UAV simulator some bugs within the OpenEaagles framework were found and published on the OpenEaagles Internet Forum. This shows that OpenEaagles is not really mature yet. The design is however solid, leveraging object-oriented principles and real-time system design techniques. By utilizing the OpenEaagles framework and incorporating JSBSim it was possible to develop a realistic and robust flight simulator within two weeks.

When searching for existing solutions for simulating flight dynamics the number of good options was lower than expected. It is still possible to rapidly develop a realistic UAV simulation with the help of existing open source projects such as FlightGear, OpenEaagles and JSBSim.

Developing a VSM is in many aspects a simpler task. The design is more straightforward and does not require advanced design patterns or complex algorithms. The VSM should simply implement the STANAG 4586 protocol; be able to receive STANAG messages, and respond to them. However, because of the sheer number of STANAG 4586 messages, it may take several weeks to implement and test the VSM, even when using an existing STANAG SDK.

It is suitable to implement a VSM in C# because of the languages flexible event handler system, and other high level functionality. The event handlers can be set up to receive and respond to STANAG messages. Classes are needed that represent CUCS, VSM and STANAG messages. During implementation of the VSM it became clear that classes for handling mission data, and field configuration data was also needed.
**Questions at Issue Answered**

1. The major components needed to rapidly develop a realistic UAV simulation are flight dynamics model, aircraft model, autopilot, and VSM.
2. The method estimated to be best suitable for rapidly developing a realistic UAV simulation is to build a modular system with the help of an existing simulation framework/flight simulator, and an existing flight dynamics model. Prototyping was successfully used as the process method during this project’s development of a UAV simulator. Other process methods might work just as well depending on the components being used, and the developers level of expertise on these components.
3. The design of a realistic UAV simulation must incorporate real-time techniques to meet human interaction requirements. An object-oriented design is recommendable to provide several layers of fidelity, and to make the simulator scalable and flexible.
4. It is possible to integrate the open source flight simulator FlightGear with a UAV simulation project. This would provide a solid flight simulation module, as well as good autopilots and several realistic flight dynamics models. It is also possible to only integrate any of FlightGear’s flight dynamics models, although most of these lack good documentation. Alternatively the simulation framework OpenEaagles can be integrated with a UAV simulation together with a flight dynamics model.
5. Because of its realism, flexibility, LGPL license and extensive documentation JSBSim is the best option for integrating an existing flight dynamics model into a UAV simulation project.

**Future Work**

To make the simulator as realistic as possible there are a few critical problems that needs to be solved:

1. Accurate aircraft models for the aircraft types that are to be simulated must be provided to JSBSim.
2. The UAV simulator lacks a good autopilot. The autopilot should have a smooth controlling mechanism for commanding heading, altitude, velocity and roll rate.
3. Being able to set commanded turn rate and pitch rate in Flight Director mode. With a good autopilot, this could easily be implemented.
4. Implementation of a circular loiter pattern. This could also be done rather easily with a good autopilot by setting a fly-to point on the circles perimeter and setting a commanded roll rate upon arrival at the fly-to point.

The UAV simulator can also be extended with a number of improvements and added functionality. These include:

- Functionality for landing and taking off from the ground.
- Controlling simulated payloads such as sensors and weapons.
- Adding more players to the simulation, including friendly and enemy air vehicles and ground vehicles. OpenEaagles provides a good basis for developing such functionality.
- Implementation of flying in Follow the Lead mode (the autopilot navigates after another “lead” player). This is rather easily implemented in the OpenEaagles framework.
• Implementation of a number of non-critical STANAG 4586 messages that are not yet implemented.
Appendix A: Sources


Appendix B: Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AP</td>
<td>Autopilot</td>
</tr>
<tr>
<td>AV</td>
<td>Air Vehicle</td>
</tr>
<tr>
<td>CUCS</td>
<td>Core UAV Control System</td>
</tr>
<tr>
<td>DLI</td>
<td>Data Link Interface</td>
</tr>
<tr>
<td>FDM</td>
<td>Flight Dynamics Model</td>
</tr>
<tr>
<td>ICS</td>
<td>Instrument Control Sweden</td>
</tr>
<tr>
<td>LOI</td>
<td>Level of Interoperability</td>
</tr>
<tr>
<td>NATO</td>
<td>North Atlantic Treaty Organization</td>
</tr>
<tr>
<td>PID</td>
<td>Proportional, Integral, Derivative</td>
</tr>
<tr>
<td>STANAG</td>
<td>Standardization Agreement</td>
</tr>
<tr>
<td>UAV</td>
<td>Unmanned Aerial Vehicle</td>
</tr>
<tr>
<td>VSM</td>
<td>Vehicle Specific Module</td>
</tr>
</tbody>
</table>
Copyright Information

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© Jonatan Rugarn