KEMI-TORNIO UNIVERSITY OF APPLIED SCIENCES
TECHNOLOGY

Häkkinen Henri

Pointing and Tracking Aid for the Modular Radar System using the Automatic Identification System

The Bachelor's Thesis Information Technology programme
Kemi 2011

PREFACE

I would like to express my gratitude for Professor Gert Hvedstrup Jensen who originally recommended me for this project, my instructor Teppo Aalto whose assistance and guidance was invaluable as well as for the whole team at the Applied Resarch department of the Danish Defence Acquistion and Logistics Organization.

ABSTRACT

Kemi-Tornio University of Applied Sciences, Technology

Degree Programme	Information Technology
Name	Henri Häkkinen
Title	Pointing and Tracking Aid for the Modular Radar System using the Automatic Identification System
Type of Study	Bachelor's Thesis
Date	9 May 2011
Pages	39 pages + 5 appendices
Instructor	Teppo Aalto, M.Sc.
Company	Danish Defence Acquisition and Logistics Organisation
Supervisor from Company	Jesper Møller, M.Sc. EE, Danish Defence

The purpose of this thesis was to develop a Pointing and Tracking Aid software engineering project for the Danish Defence Acquisition and Logistics Organization. The project was to implement a utility that is to be used for placing the measurement area of the Modular Radar System more accurately. The radar system is used to perform scientific and other diagnostic measurements for the Applied Research department of the organization.

The application uses a system known as the Automatic Identification System to identify and locate nearby ships and other naval targets. The aim is to communicate with the AIS via its own proprietary binary protocol in order for the application to receive information regarding ships. The software is also intended to be used in conjunction with the radar's control unit which was developed by the Danish Defence.

The application was developed in accordance with modern software engineering practices by first performing a throughout system analysis followed by a software architecture modelling phase and finally ending into an implementation and testing phase. UML was used as the modelling language of the software system, C++ programming language as the implementation language and Qt as the application framework.

The application was successfully developed and taken into use by the Danish Defence team of scientists. Communication with both the AIS system and the existing radar control unit was implemented successfully and verified to function properly. Some error was detected in the computed vessel distances from the AIS and the magnitude of the error was proportional to the distance itself. This suggests that the algorithm used for computing the ship distances or one of the parameters used contain inaccuracies. Further work is needed to resolve the issue.

Keywords: MRS, AIS, DALO, C++, Qt.

TIIVISTELMÄ

Kemi-Tornion ammattikorkeakoulu, Tekniikan ala

Koulutusohjelma	Tietotekniikka
Opinnäytetyön tekijä	Henri Häkkinen
Opinnäytetyön nimi	Pointing and Tracking Aid for the Modular Radar System using the Automatic Identification System
Työn laji	Opinnäytetyö
Päiväys	9.5.2011
Sivumäärä	39 + 5 liitettä
Opinnäytetyön ohjaaja	FM Teppo Aalto
Yritys	Tanskan puolustusvoimat
Yrityksen yhteyshenkilö	FM Jesper Møller

Insinöörityön aiheena oli toteuttaa Pointing and Tracking Aid -ohjelmistoprojekti Tanskan puolustusvoimien hankinta ja logistiikkayksikön käyttöön. Projektissa toteutettiin apuohjelma, jota on tarkoitus käyttää avuksi Tanskan puolustusvoimien modulaarisen tutkajärjestelmän (engl. Modular Radar System) mittausalueen tarkempaan kohdentamiseen. Kyseistä tutkajärjestelmää käytetään tieteellisten mittauksien ja muunlaisten kokeiden toteuttamiseen.

Apuohjelma käyttää avukseen Automatic Identification System -järjestelmää, joka kykenee paikallistamaan ja tunnistamaan tutka-alueen sisäpuolella kulkevat laivat sekä muut kohteet. Apuohjelma kommunikoi järjestelmän kanssa sen omalla binaarisella protokollallaan saaden tietoa lähialueen kohteista. Ohjelman on myös tarkoitus toimia yhdessä jo olemassa olevan puolustusvoimien tutkan hallintayksikön ohjelmiston kanssa.

Ohjelma kehitettiin käyttäen nykyaikaisia ohjelmistotuotannon käytäntöjä aloittaen kokonaisvaltaisella järjestelmäarkkitehtuurin analysoinnilla, edeten suunnittelu- ja mallinnusvaiheeseen sekä lopulta päätyen kehitys- ja testausvaiheeseen. Ohjelman arkkitehtuurin mallinnukseen käytettiin UML-mallinnuskieltä sekä C++-ohjelmointikieltä ja Qt-kirjastoa ohjelman tekniseen toteutukseen.

Ohjelma kehitettiin onnistuneesti ja se otettiin Tanskan puolustusvoimien tutkintayksikön käyttöön. Kommunikointi AIS-järjestelmän ja hallintayksikön ohjelmiston kanssa toteutetiin onnistuneesti ja sen toiminta varmistettiin. Laivojen lasketuissa etäisyyksissä huomattiin pieni laskennallinen virhe, jonka suuruus oli verrannollinen laivan etäisyyteen AIS-vastaanottimesta. Tämä antaa aihetta olettaa, että ohjelman laivojen etäisyyden laskemiseen käytetyssä algoritmissa tai jossain sen parametreissa on epätarkkuutta. Lisätutkimuksia tarvitaan tämän ongelman selvittämiseen.

Avainsanat: MRS, AIS, DALO, C++, Qt.

TABLE OF CONTENTS

EXPLANATION OF CHARACTERS AND ABBREVIATIONS

AIS	Automatic Identification System
API	Application Programming Interface
DALO	Danish Defence Acquisition and Logistics Organization
IP	Internet Protocol
IPC	Inter-Process Communication
ISO	International Organization for Standardization
JNI	Java Native Interface
LAN	Local Area Network
MMSI	Maritime Mobile Service Identifier
MRS	Modular Radar System
PATA	Pointing and Tracking Aid
TCP	Transmission Control Protocl
UML	Unified Modelling Language
VTS	Vessel Traffic Services

1. INTRODUCTION

The study presents a solution for a pointing and tracking software system that is used as an aid with the Modular Radar System for positioning the radar measuring area more precisely for a specific ship of interest. The Modular Radar System is in use by the Danish Defence Acquisition and Logistics Organization for performing scientific measurements and tests on vessels and other naval targets in various of military applications.

The study was conducted for a team of Danish scientists working in the Applied Research department of the Danish Defence Acquisition and Logistics Organization. During the course of the study, a working software application was designed and implemented with the team of scientists acting as an expert advisory group into the radar system for which the tracking software was targeted for.

This thesis describes the system architecture of the problem domain in question, provides a throughout system analysis and explains the implementation of the software system which was developed. The software system was developed in accordance with modern Software Engineering Principles with the Unified Process model used a basis for the design practice but without following it pedantically. Since the development team consisted of only a single programmer, the actual software engineering practice in the end was more a kin to the Agile development methodology than anything else.

Organization

The Danish Defence Acquisition and Logistics Organization is the specialized material centre and logistics authority of the Danish defence forces. The organization acquires, develops and phases out material capacities and ensures provisions in due time for the Danish Defence operations. The organization administers a budget of approximately 7 billion Danish Crowns of the total Danish defence budget and it is used to provide efficient support primarily to international operations and operational units. The organization also employs approximately 2400 employees and its headquarters are located in Ballerup near Copenhagen.

Problem Statement

The Module Radar System is used to measure the radar cross section of various objects such as ships and aircrafts. The radar is fitted into a 20 feet (approximately 6 meters) container with the radar front-end mounted on the top. The radar is controlled via a control panel software installed on a control PC that is located inside the container and a joystick is used to control the direction of the radar beam. In order to see where the radar is pointing at, a visual and an infrared camera is also fitted into the front-end casing and the camera output is displayed on the control panel software.

In poor weather conditions it can be hard to recognize a specific ship using only the visual and the infrared camera display. Multiple ships may be coaligned along the radar beam axis, making it harder to position the measuring area of the radar against the ship of interest. Therefore a more advanced pointing and tracking system is required to make sure the right ship is being measured.

This tracking system would have to display ships in the vicinity of the radar on a top-down radar map with the area of the radar beam superimposed over it. The aim is to provide means for the radar operator to observe information regarding targets such as its distance and heading. Provided with this information the radar operator would then be able to place the measuring area over the ship given its distance from the radar station.

A commercially available Automatic Identification System is an automated tracking system used on ships and by Vessel Traffic Services for identifying and locating vessels by electronically exchanging data with nearby ships and VTS stations. An AIS system will be used in conjuction with the tracking software to detect and identify ships. That is, the AIS system will track the movements of ships within the nearby region while the software will use this information provided to display ships on the radar map.

Previous Work

Previous work was done by Nicholas Howard in his *Pointing and Tracking Aid for the Modular Radar System (MRS) Using a Commercially Available Automatic Identification System (AIS)* Bachelor's thesis upon which this thesis is in partly based on. In his Bachelor project Nicholas Howard chose a Java-based implementation that was able to succesfully communicate with the Automatic Identification System using a serial port connection. His solution was elegant but fundamentally incomplete – the communication with the Modular Radar System was left completely unfinished due to lack of time and unresolved technical issues.

Solution

Due to technical reasons described later, a Java-based implementation is inconvenient and unoptimal. Therefore the solution presented in this thesis takes a different aproach. Instead C++ programming language and the Qt framework is used to develop the tracking system which shall be referred to as the Pointing and Tracking Aid in the rest of this thesis. Qt is a cross-platform application framework which is used to implement the graphical user interface of the application.

2. SYSTEM ANALYSIS

This chapter lays down the ground work upon which the actual implemention of the software rests upon. Overall system architecture description is followed by more precise subsystem descriptions and the interfaces which these subsystems use to inter-communicate. Finally we are left with a requirements and use case analysis that explains the framework how the Pointing and Tracking Aid application is to behave and the Mathematical basis for calculating the required data for each ship.

2.1. System Architecture

The system consists of four main subsystems. These are the Control PC, the Modular Radar System, the Automatic Identification System and the Pointing and Tracking Aid itself which will be used in a separate laptop computer. The system architecture is illustrated in the figure 1./8,9/

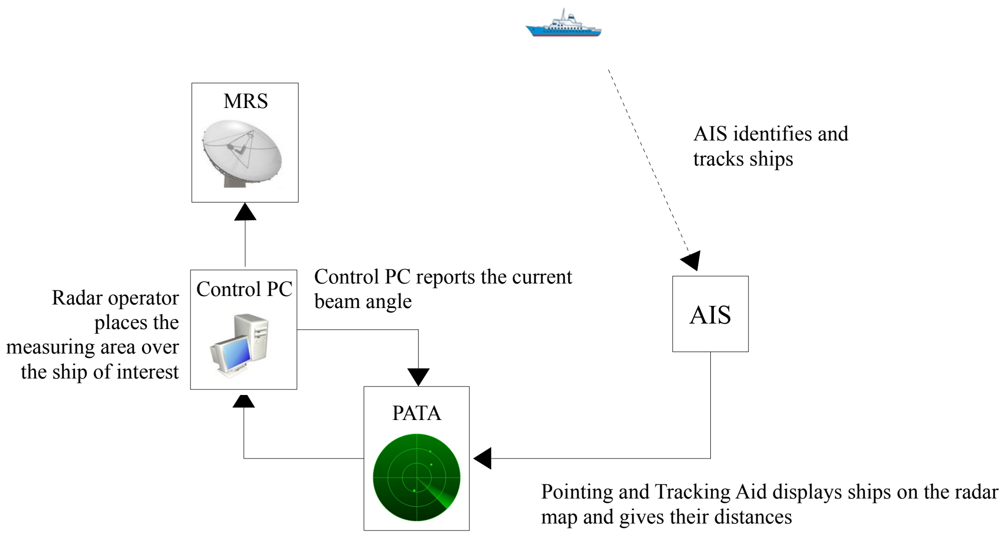

MRS
AIS identifies and tracks ships
Control PC
Control PC reports the current beam angle
Radar operator places the measuring area over the ship of interest
AIS
PATA
Pointing and Tracking Aid displays ships on the radar map and gives their distances

Fig. 1. System architecture

The Control PC contains the control panel software which is used to control the Modular Radar System. The control panel can used to rotate the radar beam horizontally and vertically, observe the visual and infrared camera output on a display and perform various diagnostic as well as scientific measurements using the radar./8,9/

The Automatic Identification System communicates with the Pointing and Tracking Aid to provide information regarding ships and other stations within its range. The subsystem reports information for each identified ship such as its geographic location, speed, course, heading, call sign, ship name, destination etc. The model that will be used in this project is the AIS 200P from Kongsberg which is only used in receiver mode./4,8,9/

The Pointing and Tracking Aid will be used from a laptop computer which is separate from the Control PC. The application communicates with the Control PC by using a Windows Mailslot to know about the current radar beam angle of the Modular Radar System. A wireless ethernet connection is used to communicate with the Automatic Identification System by using an ASCII-based NMEA 0183 protocol./8,9/

2.1.1. Modular Radar System

Inter-Process Communication mechanism known as the Windows Mailslot is used to communicate about the current MRS beam direction over the Local Area Network to each recipient who is interested to know about it. That is, each time the radar beam is reoriented to a new direction, the radar beam angle gets propagated by the control panel software to each mailslot listener./8,9/

Windows Mailslot is a Microsoft Windows operating system specific IPC mechanism which implements a one-to-many communication topology. A host opens a mailslot into which multiple others hosts may send messages. A single sending host is also able to broadcast a message to multiple mailslots over the Local Area Network without actually specifying the recipients explicitly. Mailslots are identified by path names which always have the format \\ComputerName\Mailslot\MailslotName where ComputerName is the name of the computer where the mailslot exists and the MailslotName is a unique identifier identifying a mailslot from other mailslots in the same host. A dot character may be used as the ComputerName to indicate the local computer and an asterisk character may be used to indicate all computers in the Local Area Network when a message is being broadcasted to all listening recipients. Mailslots are however limited in that they operate only within a Local Area Network – sending a message to a remote host over a Wide Area Network is not supported./5,6/

The control panel software uses a mailslot named MRS-Mail to notify changes on the radar beam direction. Each message sent contains exactly four lines. The first line always contains the string Message. The second line contains the name of the user account that is used to run the control panel software such as Administrator. The third line contains the name of the computer the control panel software runs on. The fourth line contains the current azimuth angle of the radar beam followed by the elevation angle. Both angles are expressed in degrees and are separated by a semicolon character. Lines are separated by using the Windows operating system line ending convention, that is, a carriage return followed by a line feed character. The azimuth angle is always expressed using three digits for the integer part and two digits for the fractional part while conversely two digits are used for both the integer part and the fractional part in the elevation angle. A comma is used as the decimal point. The reference angle for the radar beam azimuth points to North and the angle increases in clockwise direction while the reference angle for the elevation is coplanar with Earth's surface with the positive angle increasing upwards./8,9/

The following gives an example of a typical message sent. In this example the name of the user account that is used to run the control panel software is Administrator while the name of the computer the control panel runs on is MRS2. The radar beam azimuth angle is 90.23 degrees (pointing approximately to East) and the elevation angle is 4.01 degrees (pointing slightly upwards).

```
Message
Administrator
MRS2
090,23; 04,01
```

2.1.2. Automatic Identification System

The specific Automatic Identification System model used allows two different ways to communicate with the Pointing and Tracking Aid: a wireless ethernet connection or a serial port RS-232 connection. In the wireless ethernet case the AIS basically acts as a TCP/IP server into which clients connect to./4,8,9/

The Automatic Identification System uses a simple ASCII-based protocol in which the data is transmitted in the form of sentences. Formally this protocol is known as NMEA 0183 and it is defined by the National Marine Electronics Association. The NMEA standard is proprietary and it sells for at least 325 USD. However the protocol has been reverse-engineered by several public sources./10,14/

In the NMEA 0183 protocol each sentence begins with a dollar sign followed by two ASCII characters to identify the talker and three ASCII characters to identify the type of the message. Next a comma-separated list of data fields follows. The last data field may be followed by an asterisk and a two-digit checksum representing a hexadecimal number. The check sum is a bitwise exclusive-OR of all characters between the dollar sign and the asterisk and it may used to validate the correctness of the transmitted message. Each sentence ends with a pair of carriage return and line feed./10,14/

AIVDM/AIVDO Sentences

The Automatic Identification System however uses a special extension to the NMEA 0183 standard in which the data is encoded using a six-bit message payload armoring. To mark the beginning of these types of sentences, an exclamation mark is used instead of a dollar sign. The character pair AI is used to identify the AIS being the talker and the message type is always either VDM or VDO. Therefore each relevant six-bit armored AIS sentence always begin either with !AIVDM or !AIVDO strings./14/

The following gives an example of an AIVDM sentence:

```
!AIVDM,1,1,,B,177KQJ5000G?tO`K>RA1wUbN0TKH,0*5C
```

The fields have the following meanings:

- Field 1 !AIVDM identifies the sentence as an AIVDM message./14/
- Field 2 is the count of the fragments in the currently accumulating message. The NMEA 0183 standard limits each sentence to 82 characters so it is sometimes required to split a message into several fragments./14/
- Field 3 is the one-based fragment number of the sentence. A sentence with both the fragment count and the fragment number as one is a complete message itself./14/
- Field 4 is a sequential message identifier for a multi-sentence messages./14/
- Field 5 is a radio channel code./14/
- Field 6 is the six-bit armored message payload./14/
- Field 7 is the number of fill bits required to pad the payload to a six-bit boundary ranging from 0 to 5. Subtracting five from this gives the number of least significant bits of the last 6-bit nibble in the payload that should be ignored./14/
- The two-digit hexadecimal sequence after the asterisk is the check sum./14/

The format for AIVDO sentences is exactly the same with the exception that AIVDM sentences are used to describe remote naval targets while AIVDO sentences describe the local AIS station itself such as its geographic location./14/

Payload Armoring

Each character in the payload represents six bits of data. To deduce the value of a single character, subtract 48 from its ASCII character value and if the result is higher than 40, subtract 8. Concatenating all six-bit nibbles with the most significant bit first will give the decoded binary presentation of the payload message./14/

For example, a payload data of ”177K” gives the following series of six-bit nibbles after decoding:

```
000001 000111 000111 011011
```

When concatenated together the nibbles will yield the following series of bytes which is the decoded payload message:

```
00000100 01110001 11011011
```

Appendix 1 lists the corresponding values for each armored payload character.

Payload Datatypes

Data inside a payload message after decoding the armoring are encoded using bit fields. Bit fields are interpreted as different data types depending on the message which the payload contains./14/

The following data types are supported:

- Numeric bit fields encoded as a big endian two's-complement integers; the sign bit is the highest bit if the integer is signed./14/
- Single bit boolean flag with true encoded as a 1-bit and false as a 0-bit./14/
- Character strings encoded using a special text encoding in which each six-bit nibble maps to a specific ASCII character. Values 0-31 map to ASCII characters '@' through '_' while values 32-63 map to ASCII characters ' ' (that is, a space) through '?'. Any other ASCII character cannot be encoded. Terminating '@' marks the end of the string and that character should not be considered as being part of the text./14/

For example given the following series of six-bit nibbles:

```
001000 000101 001100 001100 001111 000000 000000 000000
```

Gives the following series of ASCII characters after decoding using the text encoding described above:

```
HELLO@@@
```

Since the @ character is used to mark the end of the text, the trailing three characters are not considered to be part of the decoded text.

Appendix 2 gives the full AIS text encoding to ASCII mapping in a tabular format.

Payload Messages

The first 6 bits of the decoded payload marks the type of the message. In total there are 27 different types of messages in the standard encoding such information as position reports, class of the ship and its voyage, navigational support aids, time and date inquiries and responses, binary encoded broadcast messages, safety related broadcasts and acknowledgements etc./14/

The Pointing and Tracking Aid however is only interested to know about message types 1-3 and 5. Message types 1-3 give information about the position, speed and course of vessels and the message type 5 contain the call sign and ship name information.

In summary, message types 1-3 contains the following fields which are of interest to the Pointing and Tracking Aid application:

1. Bits 8-37: Unique identifier of the ship encoded as an unsigned integer./14/
2. Bits 38-41: Navigation status such as “under way using engine”, “at anchor” etc. encoded as an unsigned integer code./14/
3. Bits 42-49: Rate of turn expressed in degrees per minute encoded as a signed integer./14/
4. Bits 50-59: Speed over ground expressed in knots encoded as an unsigned integer./14/
5. Bits 61-88: Longitude of the ship expressed in 1/10000th of a minute encoded as a signed integer./14/
6. Bits 89-115: Latitude of the ship expressed in 1/10000th of a minute encoded as a signed integer./14/
7. Bits 116-127: Course over ground expressed in degrees encoded as an unsigned integer./14/

Correspondingly messages of type 5 has the following fields of interest:

1. Bits 8-37: Unique identifier of the ship encoded as an unsigned integer./14/
2. Bits 40-69: Call sign of the ship encoded as a character string./14/
3. Bits 112-231: Name of the ship encoded as a character string./14/

Appendix 3 gives the full list of message types supported by the protocol and the exact encoding of the message types of interest.

2.2. Requirements Analysis

Functional requirements for the Pointing and Tracking Aid application is given in the following:

1. The application must interface with the Automatic Identification System to identify ships./8,9/
2. The application must display the identified ships on a graphical radar map as well on a list./8,9/
3. The application must provide means for the radar operator to inspect ship details which the Automatic Identification System provides. Specifically the following data must be present for each ship:/8,9/
   - Maritime Mobile Service Identifier/8,9/
   - Call sign/8,9/
   - Ship name/8,9/
   - Navigation status/8,9/
   - Rate of turn/8,9/
   - Speed over ground/8,9/

    - Course over ground/8,9/
    - Longitude/8,9/
    - Latitude/8,9/
4. The application must interface with the Modular Radar System to be informed about the current beam direction./8,9/
5. The application must display the Modular Radar System radar beam on the radar map./8,9/
6. The application must provide means for the radar operator to select a ship./8,9/
7. The application must display a direction towards the selected ship to allow the radar operator to point the Modular Radar System beam towards it./8,9/
8. The application must allow the radar operator to tag ships for logging which will subsequently cause the ship to be logged to an external log file when its state is updated from the AIS./8,9/
9. The application must display the distances, bearings and aspects between a ship and the AIS station./8,9/

Additionally the application must be operable on the Control PC which has Windows XP operating system installed./8,9/

2.3. Use Case Analysis

Three different actors exists which interact with the Pointing and Tracking Aid application. These are:

1. The Radar Operator in terms of interacting with the application's user interface.
2. The Automatic Identification System in terms of sending information regarding identified ships within its range.
3. The Modular Radar System in terms of sending the current radar beam angle. To be precise, the actual acting entity is the control panel software installed on the Control PC which communicates the information using mailslots but for simplicity and abtraction we consider this to be the radar system itself in our discussion.

Each of the described actors interact with the application in a specific way. The Radar Operator tags ships for logging, selects a ship from the ship list to highlight it in the radar map and changes the application's settings. The Automatic Identification System communicates with the application to identify ships and the Modular Radar System communicates about the current radar beam location. Use cases and the related actors are presented graphically in the figure 2 below.

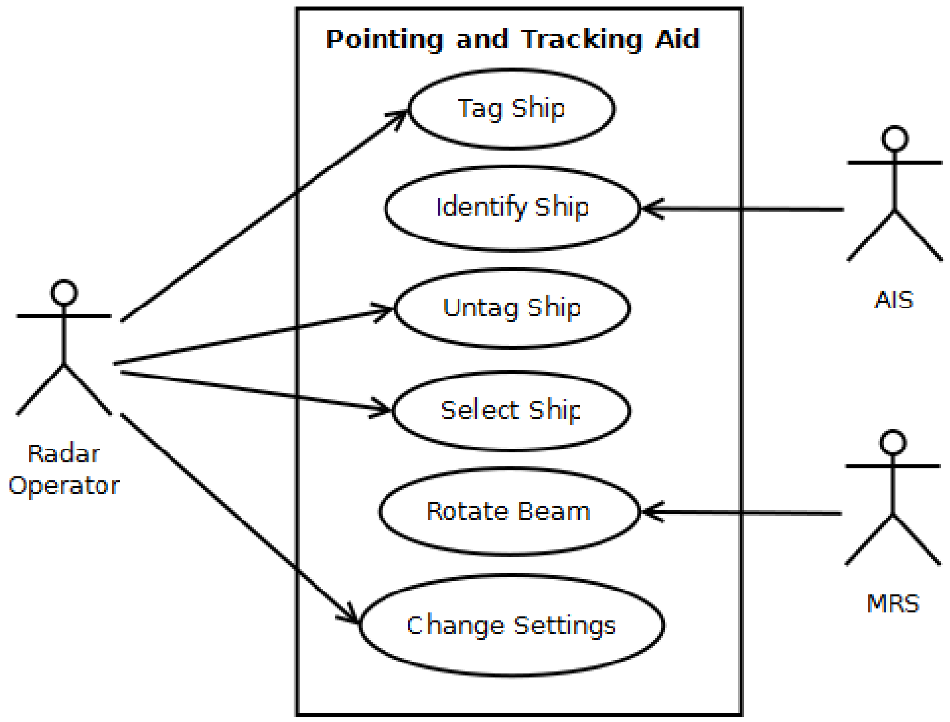


Fig. 2. Actors and use cases of the Pointing and Tracking Aid

Appendix 4 contains the full list of formal use case definitions and those are not repeated here. Instead a brief description of each interaction by each actor is given in the following section.

2.3.1. Use Case: Tag Ship

The use case begins when the Radar Operator tags a ship for logging. The ship's state is updated accordingly and any subsequent messages coming from the AIS involving the tagged ship will cause it to be logged to an external log file.

2.3.2. Use Case: Untag Ship

The use case begins when the Radar Operator untags a ship from being logged. The ship's state is updated accordingly and subsequent AIS messages involving the ship will no longer cause it to be logged.

2.3.3. Use Case: Select Ship

The use case begins when the Radar Operator selects a ship from the ship list. The Pointing and Tracking Aid updates its internal reference to the currently selected ship and redraws the relevant parts of the user interface to give a visual feedback to indicate a ship is being selected.

2.3.4. Use Case: Change Settings

The use case begins when the Radar Operator selects ”change settings” actions. The application displays a dialog which allows the settings to be changed. The changeable settings include the mailslot name which is used to listen for the Modular Radar System's current radar beam direction, the IP address and port number of the Automatic Identification System and the default longitude and latitude coordinates of the AIS receiver's location which will be used as the origin of the radar map before the AIS has sent any message to indicate its own location.

2.3.5. Use Case: Identify Ship

The use case begins when the Automatic Identification System sends a message to the Pointing and Tracking Aid regarding an identified ship or the local AIS station itself. The Pointing and Tracking Aid decodes the information contained in the message payload, updates its internal cache of ships accordingly and redraws the relevant parts of the user interface to display the ship for the Radar Operator. If a ship which the AIS message involves is tagged for logging, the ship's data is logged to an external log file.

2.3.6. Use Case: Rotate Beam

The use case begins when the Modular Radar System sends a message to the Pointing and Tracking Aid regarding a recent update in the radar beam angle. The message contains the new direction of the radar beam expressed in degrees relatively to the reference angle pointing to North. The Pointing and Tracking Aid decodes the angle from the message, update its internal state and redraw the radar map to reflect changes in the beam angle.

2.4. Mathematical Theory

There are numerous different ways of calculating distances between geographical coordinates. The Pointing and Tracking Aid adopts a method where a coordinate pair consisting of latitude φ and longitude λ is first projected on a two-dimensional x-y plane using the following equations:

$$\begin{aligned} x &= \lambda \cos \phi \\ y &= \phi \end{aligned}$$

Computations involving geographical coordinates are then conducted on a two-dimensional Cartesian coordinate space more conveniently from an algorithmical point of view than by dealing them as pairs of geographic latitude and longitude. For additional simplicity the Earth is also assumed to be a unit sphere which does introduce an error proportional to the latitude of the coordinate – as the coordinate approaches a pole the more error it should contain since the Earth is in actuality a vertically truncated ellipsoid.

The approach also does not take into consideration the cases involving geographical coordinates at opposite ends on the projected two-dimensional coordinate space – the distance drawn from a coordinate A to coordinate B as a straight line is much greater than what it actually is in reality since the Earth as a round sphere wraps around three-dimensionally.

However these considerations are negligible since we know by certainty that the problem domain deals with geographical coordinates in relative close proximity towards each other and the radar system is located near Copenhagen. We do not need to consider the coordinate space as a boundless finite space since the application will never deal with coordinates pairs at the opposite ends of the world.

2.4.1. Computing the Distance

The distance between the geographical location of the AIS and a ship is given by the Pythagora's theorem as illustrated in the figure 3.

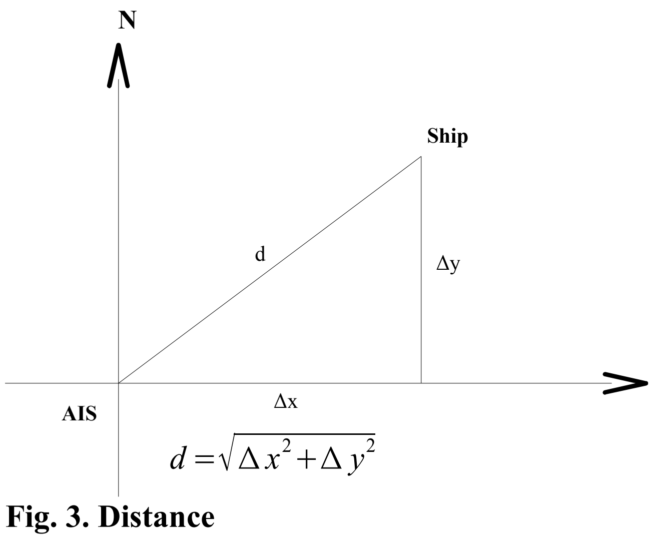


Fig. 3. Distance

The distance d is the distance between two points on a unit sphere given in radians. The actual distance on the Earth's surface is given by multiplying d by the Earth's radius.

2.4.2. Computing the Bearing

An angle between a line drawn from the coordinate system origin to a point in the two-dimensional Cartesian coordinate space and one of the coordinate system's axis can be calculated by using trigonometry as illustrated in the figure 4.

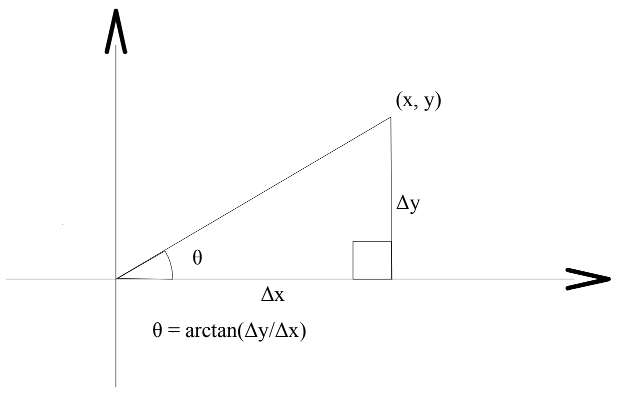


Fig. 4. Angle to the specific point of interest

However special cases must taken into account when the coordinate lies within different quadrants of the coordinate system – a constant term must be added to the equation in order to calculate the full angle relatively to the reference axis of choice. This is illustrated in the figure 5.

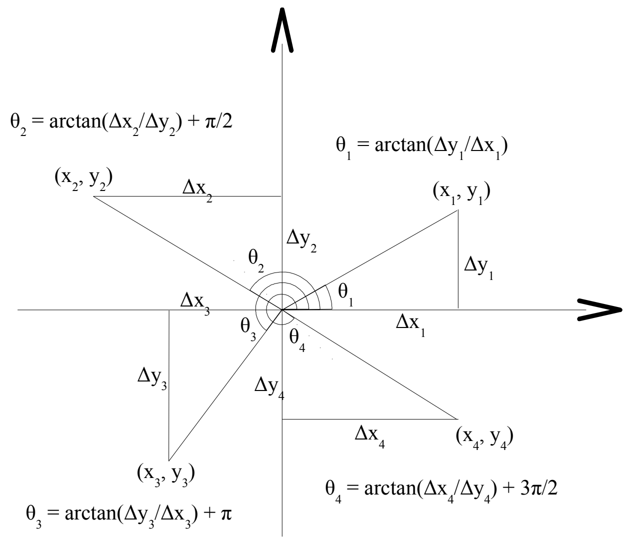


Fig. 5. Special cases in calculating the full angle

Luckily the two-argument variation of the arctangent function – atan2 in the C standard library – gives the angle in radians between the positive x-axis and a point, taking into account the coordinate system quadrant the point lies within. The arctangent function variation returns a signed angle value in the range [-π, π] which is positive for points residing in the upper half of the coordinate system and negative for points residing in the lower half of the coordinate system. This is illustrated in the figure 6./1/

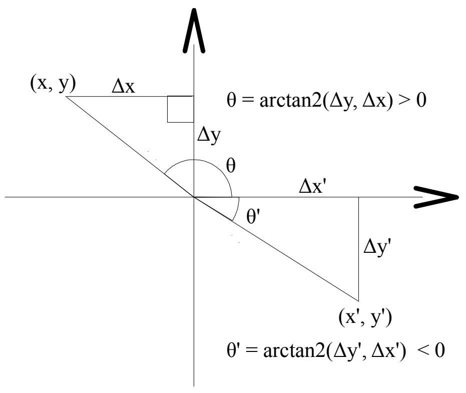


Fig. 6. Arctan2 function

Bearing is the angle between a line drawn from the geographic location of the AIS to the geographic location of a ship and the reference angle pointing to North. The angle increases in clockwise direction. This angle can be calculated using the two-argument variation of the arctangent function as illustrated in the figure 7.

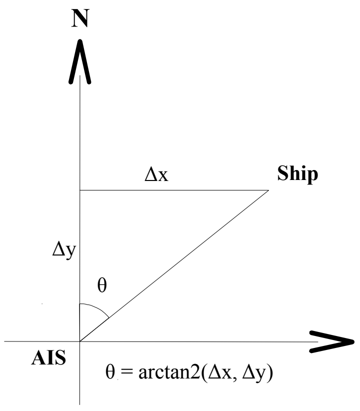


Fig. 7. Bearing angle

2.4.3. Computing the Aspect

In the context of the Pointing and Tracking Aid, an aspect is the relative direction of a ship as seen from the point of view of the radar operator. When a ship is facing directly towards the AIS, the aspect is zero. Conversely when a ship facing directly away from the AIS, the aspect is 180 degrees. The aspect increases in a counter-clockwise direction as the ship rotates as illustrated in figure 8.

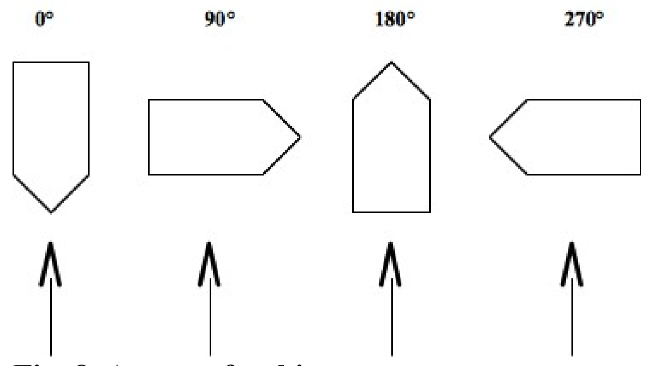


Fig. 8. Aspect of a ship

The aspect can simply be calculated as the difference between the bearing angle from the ship towards the AIS station and the course over ground of the ship. This is illustrated in the figure 9.

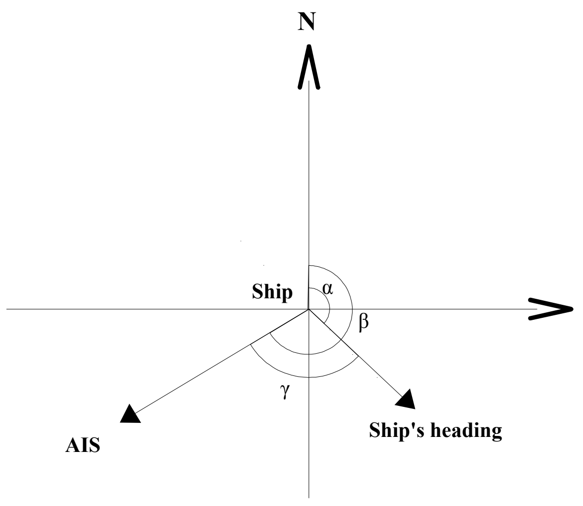


Fig. 9. Aspect angle

That is

$$\gamma = \beta - \alpha$$

Where α is the course over ground of the ship, β is the bearing angle and γ is the aspect angle.

3. DESIGN AND IMPLEMENTATION

This chapter builds on the system analysis conducted in the Chapter 2 and provides a detailed software architecture for the Pointing and Tracking Aid application. Considerations are given on choosing the programming language and the user interface toolkit for the application and a rationale is given why these tools were chosen. Finally the use cases are realized for the software architecture explained.

3.1. Considerations

The biggest issues which must be taken into consideration is the fact that the application must interface with the Control PC by using a legacy Windows Mailslots IPC mechanism. This puts some restrictions on the implementation chosen since the application must be able to call Windows specific system calls to implement this feature.

Also it should be mentioned that operating the AIS protocol does require a support for bitwise binary operations since the payload armoring needs to be decoded and the data fields extracted using bit fields in binary level.

Other minor considerations include such things as stability, continued and possibly long-term support for the user interface toolkit chosen and the possibility of porting the application to newer platforms than Windows XP.

3.1.1. Programming Language

C/C++ programming language was chosen since it compiles into native machine code, allows direct calling of Windows operating system calls and supports bitwise operations. The language itself does not contain any additonal runtime system other than perhaps the standard library and the application framework which is used to implement the graphical user interface.

The class library of the Java programming language does not contain functionality to support the Windows Mailslot IPC mechanism. This is understandable due to design priciples which Java is using – Java is designed to be portable across operating systems and platforms so it does not support platform specific features. Additonally Java does not have well formed support for manipulating data at binary level which is required to operate the AIS protocol./3/

Java does however allow arbitrary system calls to be made using additional wrapper known as Java Native Interface. This programming framework allows native applications to call Java and conversely allows Java to call native code./3/

The JNI interface however does introduce additional complexity to the application which is inherently unnecessary. Combined with the lack of bitwise operations Java is not therefore the optimal solution for implementing the Pointing and Tracking Aid.

3.1.2. Application Framework

Qt framework was chosen as the application framework for the Pointing and Tracking Aid but also the following other frameworks were considered.

- MFC (Microsoft Foundation Classes) is a C++ application framework for the Windows operating system written by Microsoft./7/
- wxWidgets (formerly known as wxWindows) is a cross-platform C++ application framework which is in use across many industry sectors including Xerox, AMD, NASA and others./15/
- Gtk+ is another application framework written in the C programming language and it is being used most notably in the desktop Linux world and was originally developed as the widget toolkit for the GNU Image Manipulation Utility (also known as GIMP)./2/

The table 1 gives a comparison over these mentioned frameworks.

Table 1. Framework comparison

Framework	Written in	License	Cross-platform
MFC	C++	Proprietary	No
wxWidgets	C++	Open source	Yes
Gtk+	C	Open source	Yes
Qt	C++	Open source	Yes

Qt is a cross-platform application framework which is widely used for developing application software with a graphical user interface. The framework is most notably used in such applications as Autodesk Maya, Google Earth, KDE desktop system, Adobe Photoshop Album, Skype, VLC media player, VirtualBox and Wolfram Mathematica. Qt was originally developed by a Norwegian company known as Trolltech but has since been acquired by the Nokia Corporation./11,12/

Qt uses standard C++ but employs a special code generator known as the Meta Object Compiler to extend the capabilities of conventional C++. These extensions include such as runtime introspection, signal and slots system and asynchronous function calls. Additionally Qt provides a fully integrated development environment called Qt Creator and an extensive class library supporting features such as SQL database access, XML parsing, unified cross-platform API for file handling, network access and thread management among many others./11/

Qt was chosen over the other choices since it has a well engineered and extensive object-oriented class library providing ready to use facilities for implementing rich graphical user interface multi-threaded applications and communicating with other processes using TCP/IP sockets. Qt provides much more convenient programming model than Gtk+ which is a more lower level framework implemented in the C programming language or wxWidgets or MFC which use preprocessor macro based message maps to route system events.

3.2. Software Architecture

This section describes the software architecture of the Pointing and Tracking Aid application. UML class and activity diagrams are used for presenting the application's static structure and runtime behaviour. Code listings are given to illustrate a behaviour more precisely. Comments are removed and only the high-level parts are shown to explain a point. Interested readers should refer to Appendix 5 for the full source code. An overview of classes and how they interact with each other is given in the figure 10.

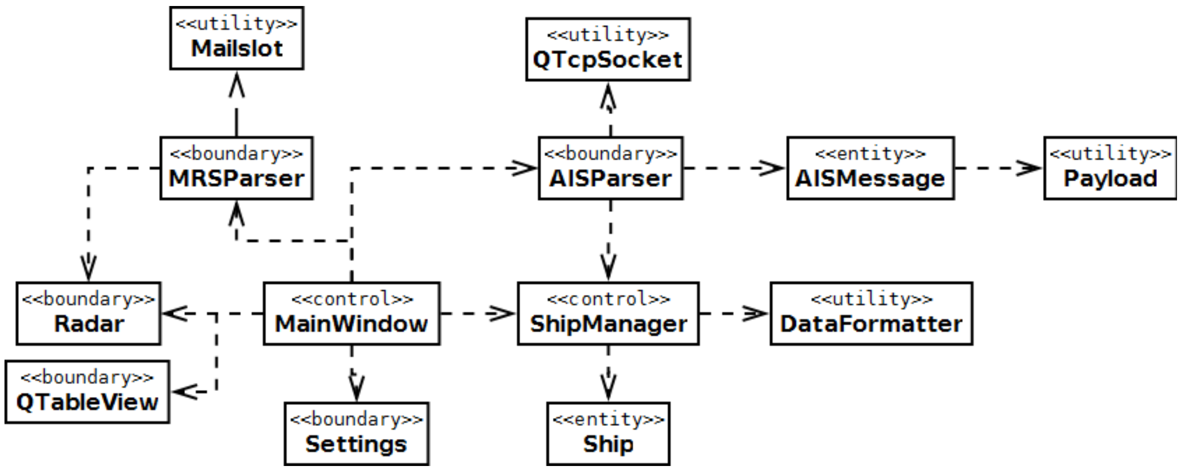


Fig. 10. Overview of classes

UML class diagrams presented in the following sections do not fully model all classes completely but omit certain information for clarity. Typically each class diagram provide enough information to discuss a certain subset of the application's behaviour rather than intending to rigorously document every service provided by a class. Such things as constructors are omitted as well as accessor methods since they are not typically relevant for discussing the application's internal logic.

The following UML stereotypes are used to model addtional behaviour:

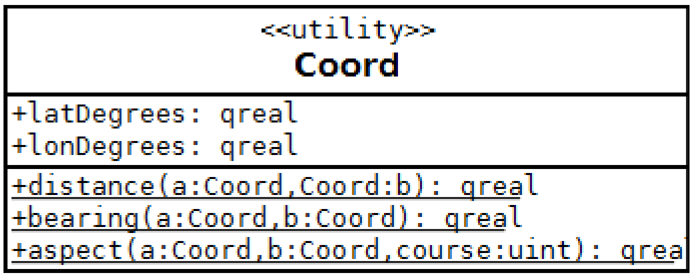
- The <<signal>> operation stereotype indicating the operation is a Qt signal.
- The <<slot>> operation stereotype indicating the operation is a Qt slot.

- The <<abstract>> class stereotype indicating the class is not intended to be instantiated (whether or not the class is actually an abstract in the strict sense).
- The <<entity>> class stereotype indicating the class encapsulates information about something while providing very little behaviour.
- The <<utility>> class stereotype indicating the class provides some, usually low-level, services on behalf of other classes while not necessary being useful alone.
- The <<boundary>> class stereotype indicating the class exists on the boundary between the application and an external actor communicating with the system.
- The <<control>> class stereotype indicating the class acting as a central coordinator of behaviour.
- The <<emits>> relationship stereotype indicating that one or more Qt signals is connected to a slot or slots of another class. An arrow indicates the signal emission direction – the arrowhead points to the class which contains the slot.
- The <<uses>> relationship stereotype indicating that a class uses another class to perform an operation.

The following naming conventions are also used throughout the application:

- Classes beginning with the character Q are part of the Qt framework and therefore are not discussed in great lengths in this thesis. Interested readers are encouraged to consult the Qt reference manual.
- Variable names beginning with an underscore indicates a private instance variables. A class usually defines a pair of accessor methods for setting and getting the value of these variable.

In the discussion which follow, one class is omitted. This is the Coord datatype class which implements a geographical coordinate and operations such as calculating the distance between two of them. This class is illustrated as an UML diagram in the figure 11.


<<utility>>
Coord
+latDegrees: qreal
+lonDegrees: qreal
+distance(a:Coord,Coord:b): qreal
+bearing(a:Coord,b:Coord): qreal
+aspect(a:Coord,b:Coord,course:uint): qrea
Fig. 11. Coord class

3.2.1. Main Window

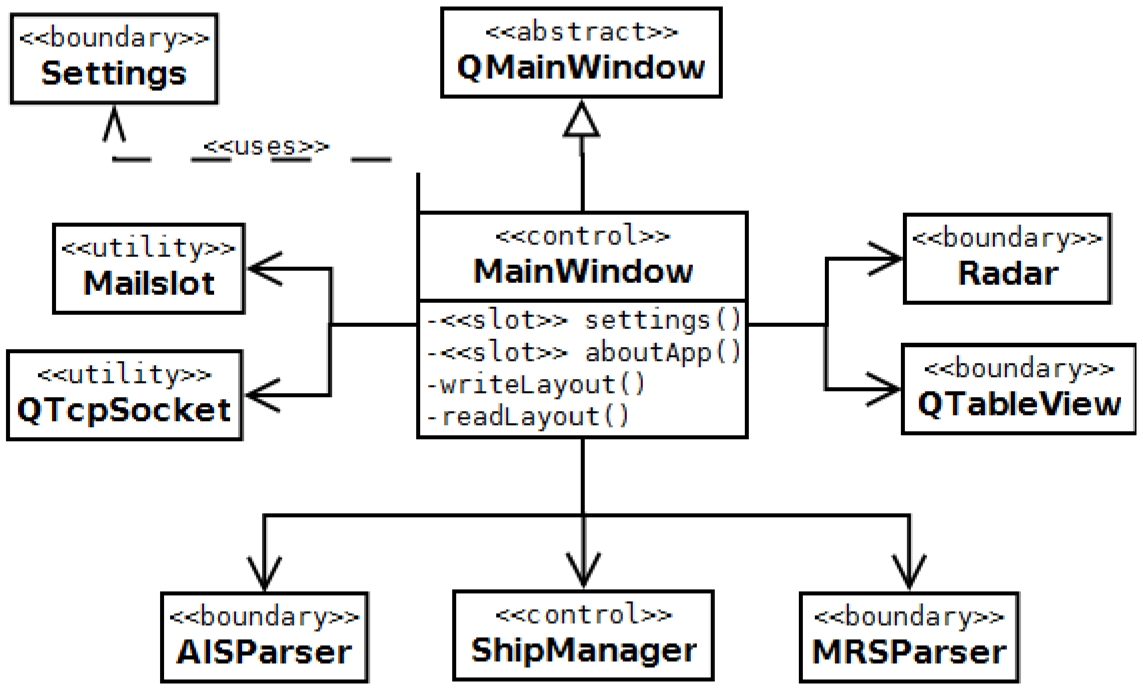


Fig. 12. MainWindow class

MainWindow as illustrated in the figure 12 is the first object created (excluding the Qt application object explained below) and it acts as a central place for setting up the rest of the application. The class is responsible for creating all the other objects and connecting signals and slots approatively. It also handles the menu actions ”change settings” and ”about the application” by connecting them to the slots settings and aboutApp respectively. The readLayout and writeLayout are called at applications startup and shutdow to save and restore the user interface layout between application runs.

MainWindow object is created in the application's main function. The following contains its code listing:

```
int main(int argc, char **argv) {
    QApplication a(argc, argv);
    QCoreApplication::setApplicationName(ApplicationName);
    QCoreApplication::setOrganizationName(OrganizationName);
    QCoreApplication::setOrganizationDomain(OrganizationDomain);

    QDir dir;
    dir.mkpath(LogDirectory);

    MainWindow w;
    w.show();

    return a.exec();
}
```

The Qt application object is created first which performs necessary initialization procedures of the application framework. The application object is given the application's and the organization's name and its domain which are used for finding the correct location in the user preferences database for storing the user interface layout and other persistent settings.

3.2.2. Settings

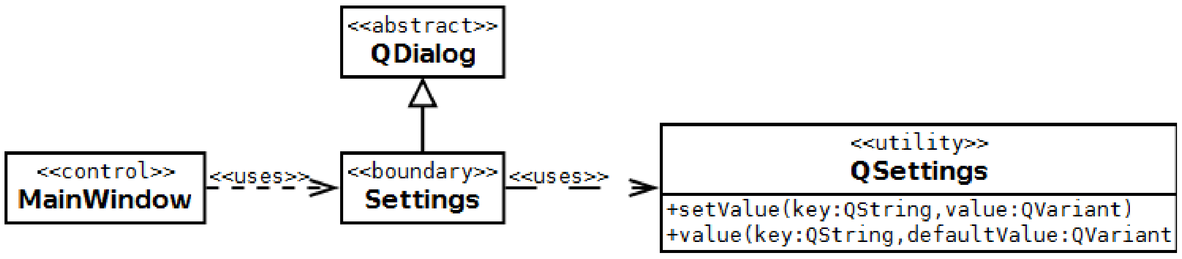
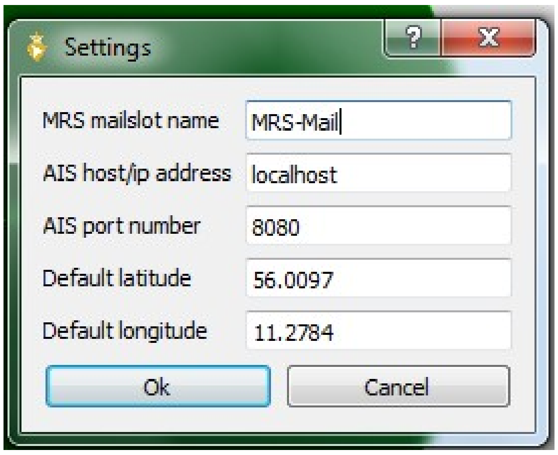


Fig. 13. Settings class

Settings class is described in the figure 13. The class is responsible for handling a modeless dialog box which is used to alter the application settings. These are:

- MRS mailslot name
- AIS IP address or host name
- AIS port
- Default latitude
- Default longitude

The class uses QSettings to actually access the operating system's registry – once the ok button is pressed, the new settings are stored persistently using it. The Settings dialog is illustrated in the figure 14.



MRS mailslot name: MRS-Mail
AIS host/ip address: localhost
AIS port number: 8080
Default latitude: 56.0097
Default longitude: 11.2784

Fig. 14. Settings dialog

3.2.3. Mailslot

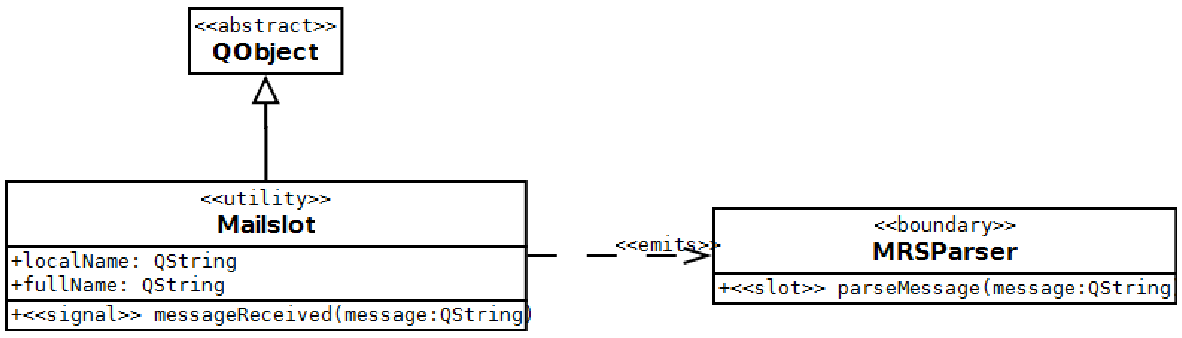


Fig. 15. Mailslot class

Mailslot as seen in the figure 15 implements an object-oriented Qt wrapper around the low-level Windows Mailslots API. Listening for incoming messages is handled asynchronously and interested parties wishing to get notified when something is received are done so by using the Qt's signal-slots mechanism. Mailslot does not by itself perform any processing or parsing on the received message but instead forwards it as is to any listening slot and relies on the receiver of the signal to make sense of it.

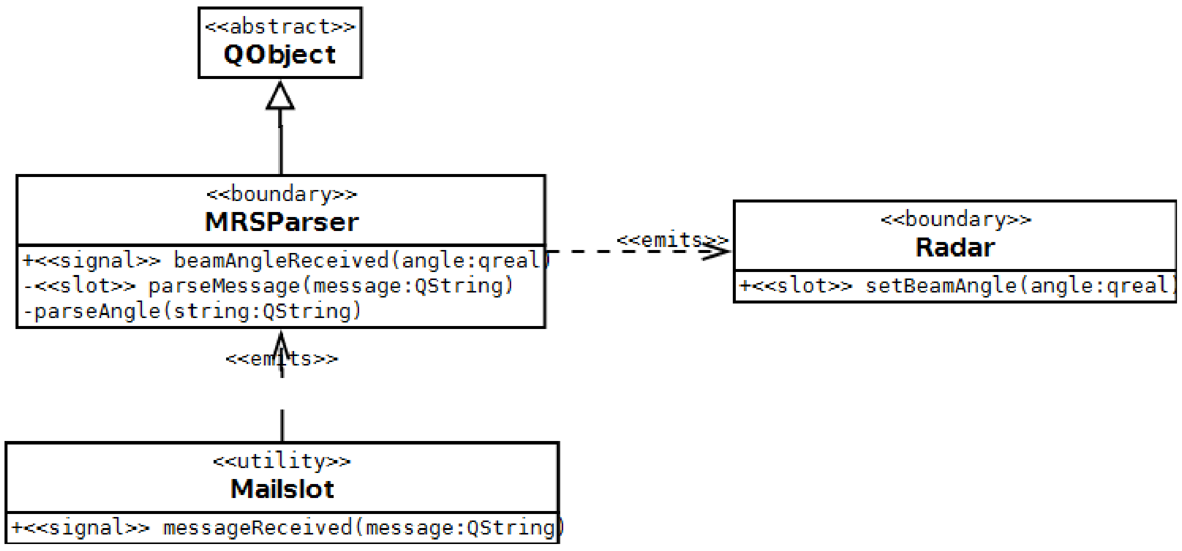
The following code listing illustrates the background poll loop which keeps reading messages coming through the mailslot:

```
void Mailslot::pollLoop() {
    while (1) {
        if (!pollMessage()) break;
    }
}

bool Mailslot::pollMessage() {
    char buffer[424];
    DWORD length = 0;
    if (!ReadFile(_win32Handle, buffer, 424, &length, 0)) {
        return false;
    }
    QByteArray message(buffer, length);
    emit messageReceived(QString(message));
    return true;
}
```

The pollLoop method is called asynchronously once the operating system level mailslot is open. The loop keeps polling for new messages until the mailslot handle is closed and in which case the ReadFile Windows system call return false. It should also be noted that the ReadFile call blocks until there is a message to receive. Once a message is available a QString object is created for it and the messageReceived signal gets emitted to whatever slot it is connected to. Qt handles the signal emission across thread boundaries by itself so there's no need to use a mutex or other thread synchronization primitives.

3.2.4. MRS Parser



<<abstract>>
QObject
<<boundary>>
MRSParser
+<<signal>> beamAngleReceived(angle:qreal)
-<<slot>> parseMessage(message:QString)
-parseAngle(string:QString)
<<emits>>
<<boundary>>
Radar
+<<slot>> setBeamAngle(angle:qreal)
<<emits>>
<<utility>>
Mailslot
+<<signal>> messageReceived(message:QString)

Fig. 16. MRSParser class

The figure 16 illustrates the MRSParser class which listens for incoming messages from the Module Radar System and parses them into radar beam angles which in turn are forwarded to the Radar. The class does not handle the low-level details of communicating with the MRS but relies on the Mailslot to implement that in any way approative.

Since the protocol which the MRS uses for communicating the radar beam angle contains other unnecessary information as well, the class uses regular expressions for extracting the interesting part of the message. What is known is the fact that the radar beam angle is located at the begining of the fourth line in the message containing three digits and a decimal point followed by two digit fractional part and terminating into a semicolon. To make the parser a little more permissive – just in case the message format changes slightly in the future – the radar beam is allowed to be located at any line and the decimal point may either be a comma or a dot. The radar beam angle is also not required to be located at the beginning of a line. Therefore the following regular expression is used:

```
[0-9]{3}[,.][0-9]{2};
```

That is, three digits in the range 0-9 followed by a comma or a dot as the decimal point followed by two digits in the range 0-9 and terminating into a semicolon. The regular expression matches the first string found conforming with this format. Everything else in the message is ignored.

3.2.5. AIS Parser

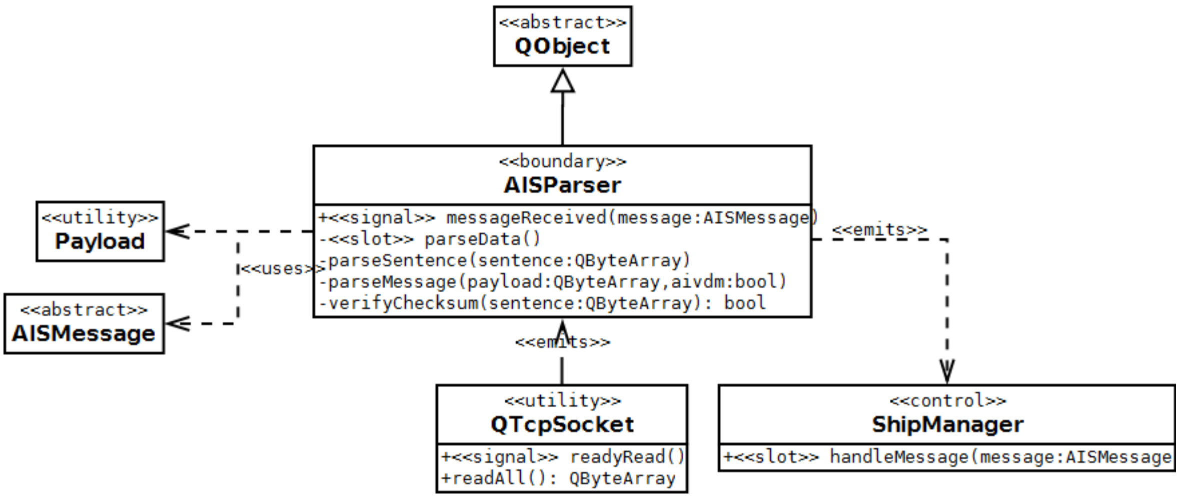


Fig. 17. AISParser class

AISParser as seen in the figure 17 listens for incoming AIVDM/AIVDO sentences from the AIS system and parses them into something meaninful. The class relies on a QTcpSocket to handle the low-level details of communicating with the AIS at the TCP/IP level.

Since the protocol is rather complicated, the responsibility of parsing it is split between the classes AISParser, Payload and AISMessage derived classes. AISParser class handles the parsing the protocol at a sentence level by splitting the sentence into fields, extracting the payload and handling multi-fragmented sentences. Payload class handles decoding the payload armoring and AISMessage derived classes each handle a specific message class.

3.2.6. AIS Messages

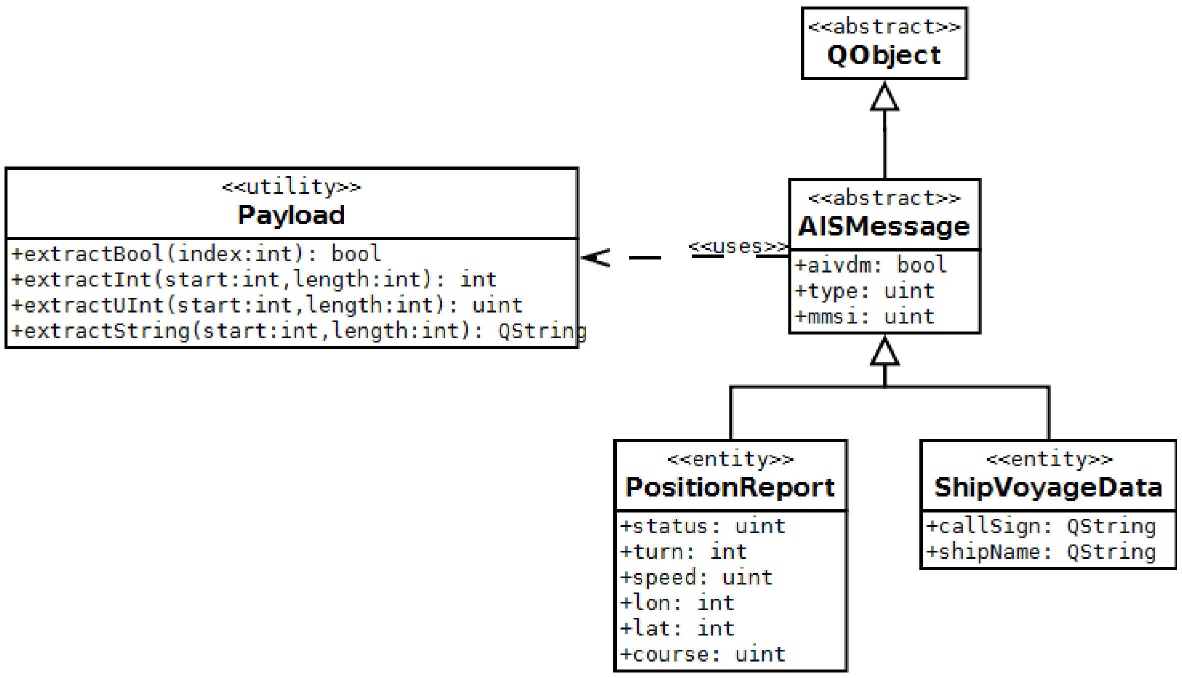


Fig. 18. AIS message classes

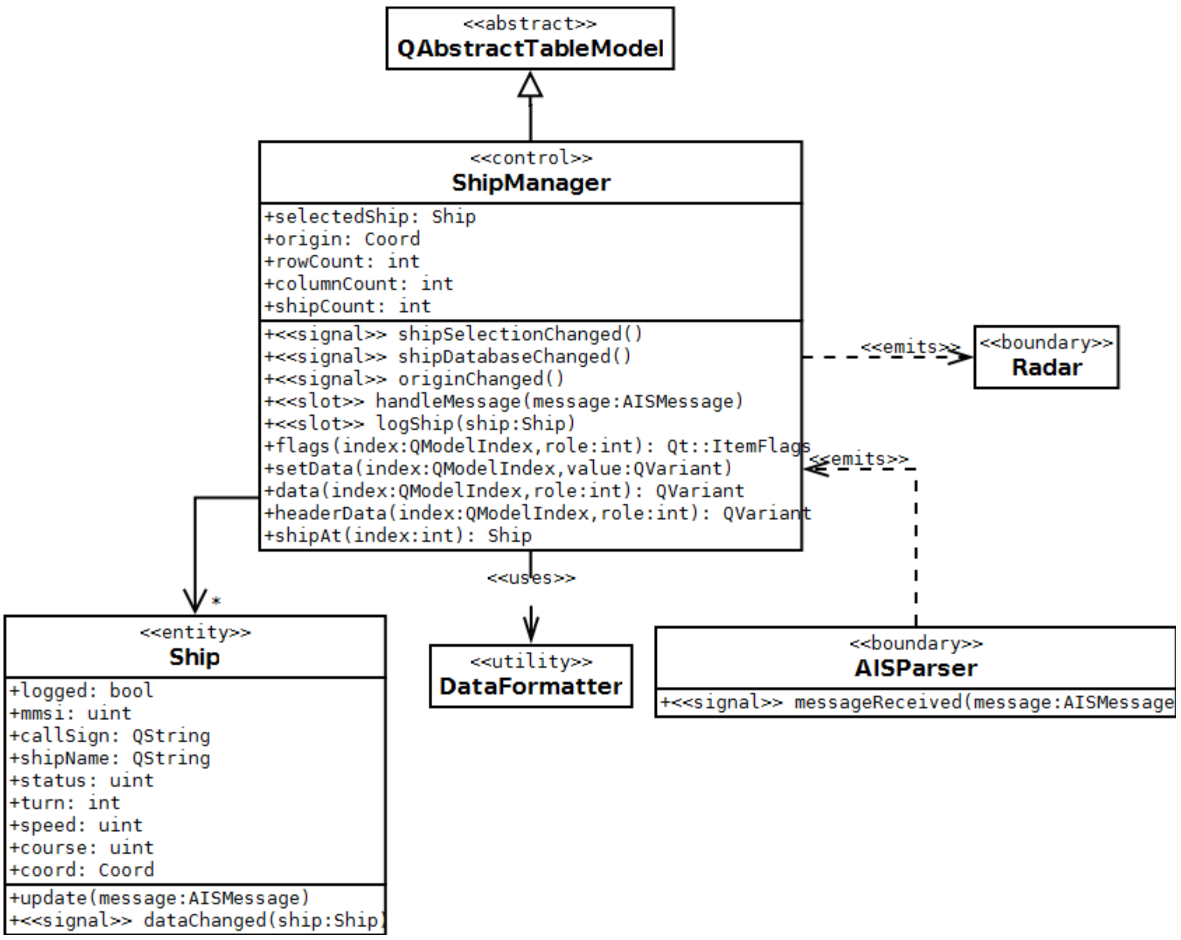
The figure 18 illustrates AIS message classes which are used to parse a specific kind of message received from the AIS.

AISMessage is the abstract base class for all AIS message types supported by the Pointing and Tracking Aid application. AISMessage itself carries only the type identifier of the actual message, a Maritime Mobile Service Identifier that identifies the ship for which the message was received and a flag indicating whether the message sentence was AIVDM or AIVDO. Derived classes define other members which are relevant to the message type in question.

Two AISMessage derived classes are defined: PositionReport which corresponds to the message types 1-3 and ShipVoyageData which corresponds to the message type 5.

Payload helper class is used to extract data fields from the armored message payload. The class contains methods for extracting each data type supported, all of which take a range of bit fields to extract the data from.

3.2.7. Ship Manager



<<abstract>>
QAbstractTableModel
<<control>>
ShipManager
+selectedShip: Ship
+origin: Coord
+rowCount: int
+columnCount: int
+shipCount: int
+<<signal>> shipSelectionChanged()
+<<signal>> shipDatabaseChanged()
+<<signal>> originChanged()
+<<slot>> handleMessage(message:AISMessage)
+<<slot>> logShip(ship:Ship)
+flags(index:QModelIndex,role:int): Qt::ItemFlags
+setData(index:QModelIndex,value:QVariant)
+data(index:QModelIndex,role:int): QVariant
+headerData(index:QModelIndex,role:int): QVariant
+shipAt(index:int): Ship
<<emits>>
<<boundary>>
Radar
<<emits>>
<<uses>>
*
<<entity>>
Ship
+logged: bool
+mmsi: uint
+callSign: QString
+shipName: QString
+status: uint
+turn: int
+speed: uint
+course: uint
+coord: Coord
+update(message:AISMessage)
+<<signal>> dataChanged(ship:Ship)
<<utility>>
DataFormatter
<<boundary>>
AISParser
+<<signal>> messageReceived(message:AISMessage

Fig. 19. ShipManager class

ShipManager as given in the figure 19 is the repository for storing identified ships in the application's memory. It maintains a list of ships identified so far by the Automatic Identification System and provides means for accessing and searching them. The class is also derived from the QAbstractTableModel base class which means that it can be used as the data model for table-based Qt user interface objects such as the QTableView. The handleMessage slot takes in an AISMessage object which is used to to keep the internal list of ship up to date.

Ship class encapsulates data for each identified ship. A Ship knows how to update itself given a AISMessage object encoding data received from the AIS but it doesn't do anything else. Ships can be flagged for being logged and when the flag is set, the update method emits dataChanged signal each the time the ship's data changes. This signal is connected to ShipManager's logShip slot which performs the actual logging procedure.

DataFormatter helper class is used for formatting raw data received from the AISParser into a human readable form. The class contains methods at the class level for each class of data but no methods at the instance level.

3.2.8. Radar

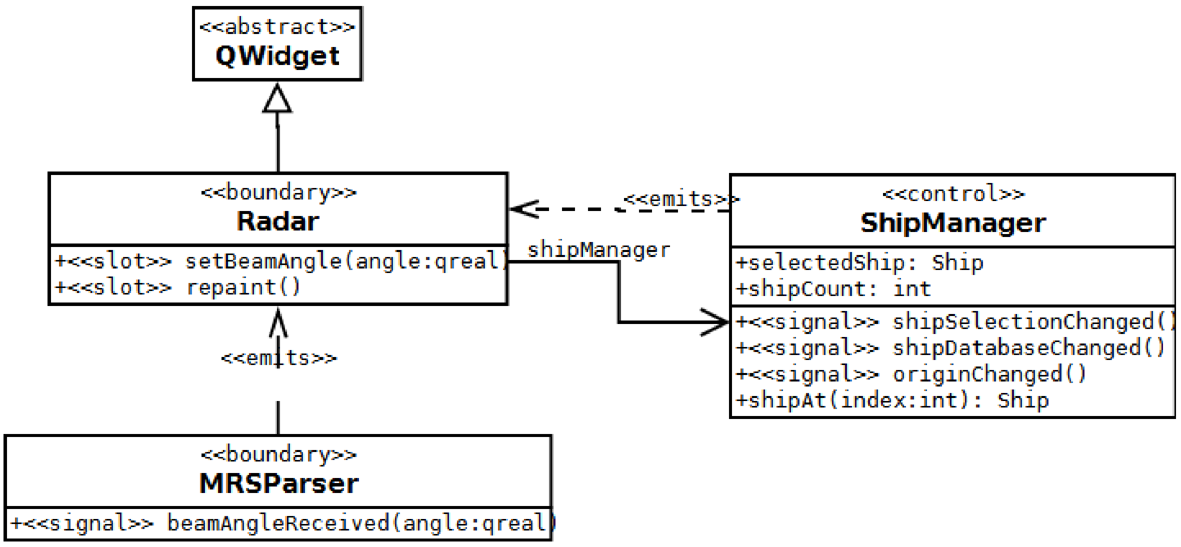


Fig. 20. Radar class

The figure 20 describes the Radar class which is the user interface element responsible for drawing the radar map with the area of the radar beam superimposed on top of it. Ships are drawn as small triangular shapes as illustrated in the figure 21.

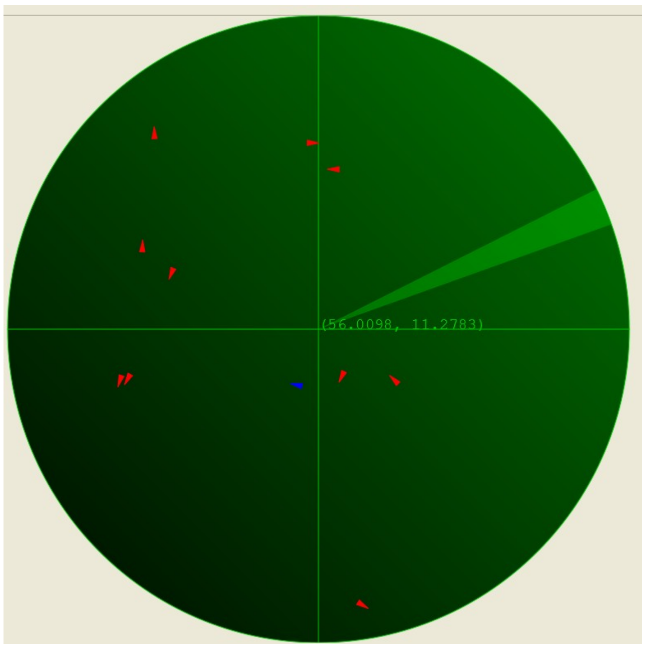
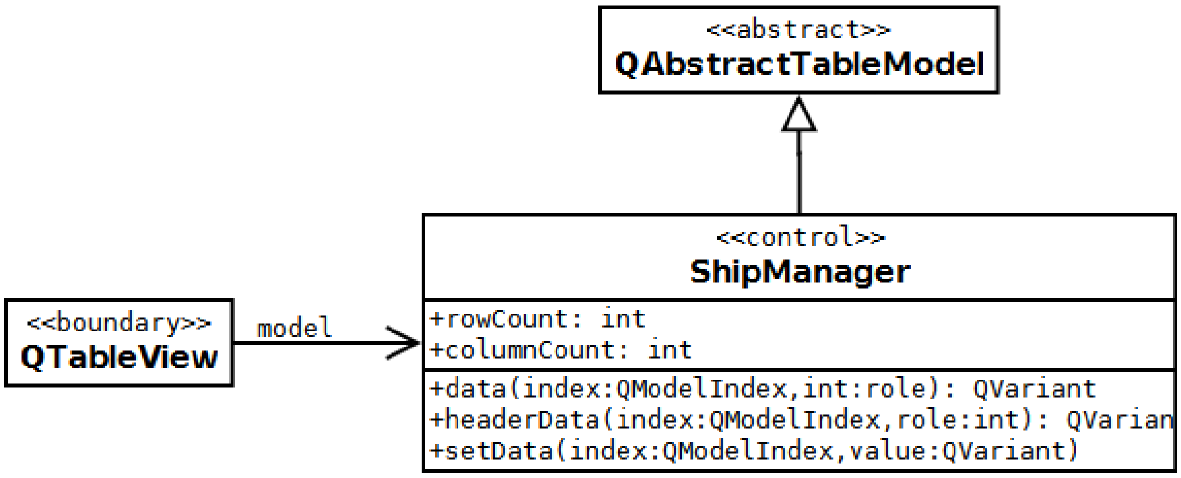


Fig. 21. Radar map showing ships

Radar uses the ShipManager to know about the locations of ships and the current coordinate system origin. The setBeamAngle slot is connected to the MRSParser's beamAngleReceived signal to keep the radar up to date about the changes of the radar beam direction while the repaint slot is called by the shipSelectionChanged, shipDatabaseChanged and originChanged signals of the AISParser to reflect the changes on the state of the system on the radar map.

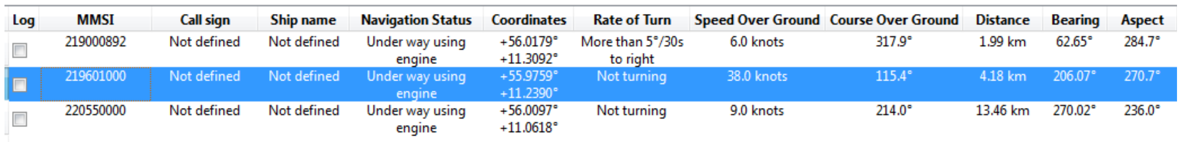
3.2.9. Ship List



<<abstract>>
QAbstractTableModel
<<control>>
ShipManager
+rowCount: int
+columnCount: int
+data(index:QModelIndex,int:role): QVariant
+headerData(index:QModelIndex,role:int): QVarian
+setData(index:QModelIndex,value:QVariant)
<<boundary>>
QTableView
model

Fig. 22. Classes involving the ship list

The figure 22 illustrates the ship list. The ship list is the part of the user interface which displays ships on a tabular form as seen in the figure 23.



Log	MMSI	Call sign	Ship name	Navigation Status	Coordinates	Rate of Turn	Speed Over Ground	Course Over Ground	Distance	Bearing	Aspect
☐	219000892	Not defined	Not defined	Under way using engine	+56.0179° +11.3092°	More than 5°/30s to right	6.0 knots	317.9°	1.99 km	62.65°	284.7°
☐	219601000	Not defined	Not defined	Under way using engine	+55.9759° +11.2390°	Not turning	38.0 knots	115.4°	4.18 km	206.07°	270.7°
☐	220550000	Not defined	Not defined	Under way using engine	+56.0097° +11.0618°	Not turning	9.0 knots	214.0°	13.46 km	270.02°	236.0°

Fig. 23. Ship list

Since the ShipManager class derives from the QAbstractTableModel class and Qt provides a ready-made QTableView class which suits for this purpose, no derived class is needed. The ship list is basically just an instance of the QTableView class and the ShipManager is set as its model. Each time the ShipManager updates its internal list of ships, the QTableView instance gets updated automatically by the Qt framework. QTableView class also provides functionality for handling ship selection.

3.3. Use Case Realization

This section realizes the use cases which were presented in the previous chapter and more formally in the Appendix 4. Realization involves taking each use case and implementing them in terms of the software architecture presented in the previous section.

3.3.1. Use Case: Tag Ship

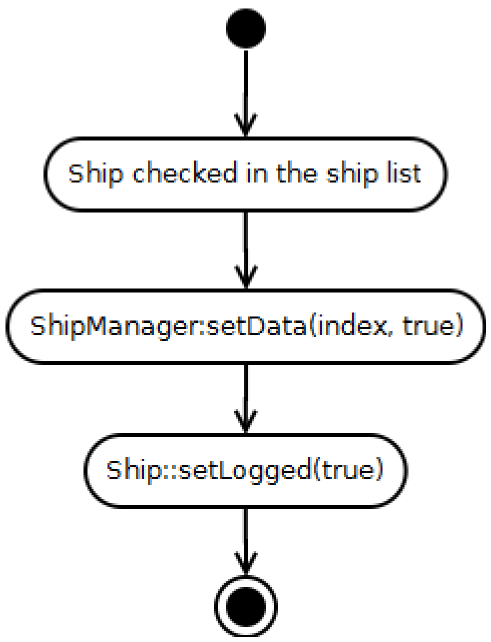

Ship checked in the ship list
ShipManager:setData(index, true)
Ship::setLogged(true)

Fig. 24. Tag Ship use case realization

Tag Ship use case as seen in the figure 24 handles the tagging of ships for logging. The ship list contains a check box for each ship and when the ship is checked in this way, the ship will be logged when its state changes as a response to a message received from the AIS.

The use case begins when the Radar Operator checks a ship in the ship list. This calls the setData method for the ShipManager class for the specified index corresponding to the ship's location in the ship list. The method searches the ship from the list by using the index and calls the setLogged method for the Ship with a value corresponding to true. This will set the logging flag on for the ship and therefore making the ship to be logged next time its state changes.

3.3.2. Use Case: Untag Ship

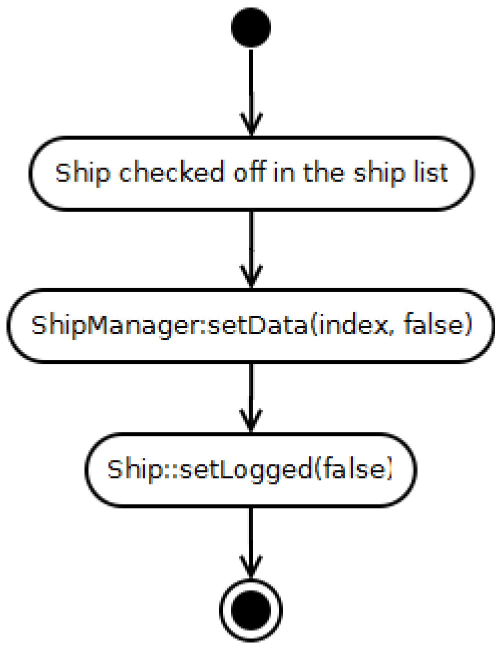

Ship checked off in the ship list
ShipManager:setData(index, false)
Ship::setLogged(false)

Fig. 25. Untag Ship use case realization

Untag Ship use case seen in the figure 25 is similar to Tag Ship use case.

The use case begins when the Radar Operator checks off a ship in the ship list. This causes the setData method of the ShipManager to be called with the ship's index passed in as the argument. The ship is searched from the database using the provided index and the logging flag is turned off by calling setLogged method for the ship with false as the argument.

3.3.3. Use Case: Select Ship

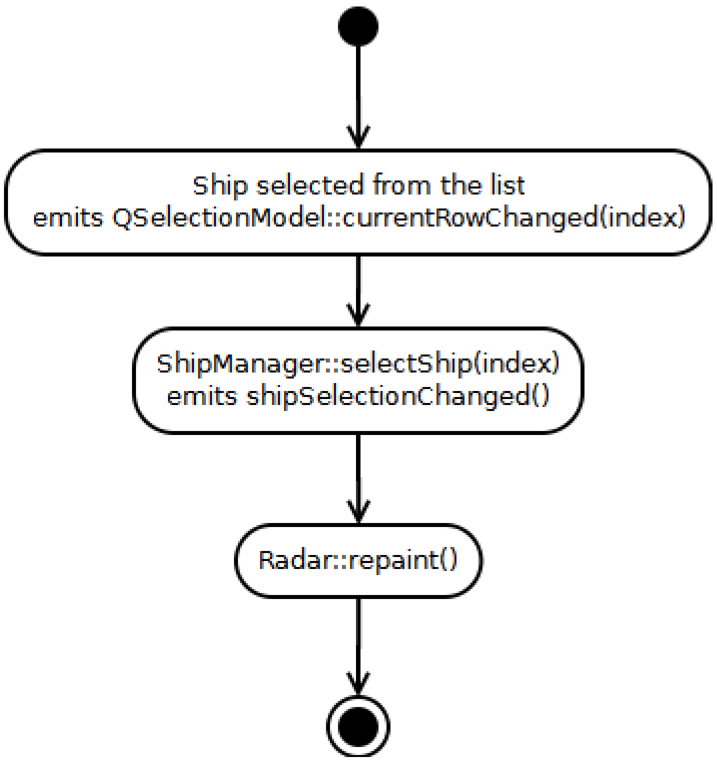

Ship selected from the list
emits QSelectionModel::currentRowChanged(index)
ShipManager::selectShip(index)
emits shipSelectionChanged()
Radar::repaint()

Fig. 26. Select Ship use case realization

The figure 26 illustrates the Select Ship use case which handles ship selection from the ship list. Selecting a ship from the ship list causes the Radar to update itself to highlight the currently selected ship.

The use case begins when the Radar Operator selects a ship from the ship list. This will cause the currentRowChanged signal of the QTableView's selection model (the QSelectionModel class) to get emitted which in turn is connected to the selectShip slot of the ShipManager object. ShipManager updates the currently selected ship reference and emits the originChanged signal. The signal is connected to the repaint slot of the Radar object which redraws the radar map.

3.3.4. Use Case: Change Settings

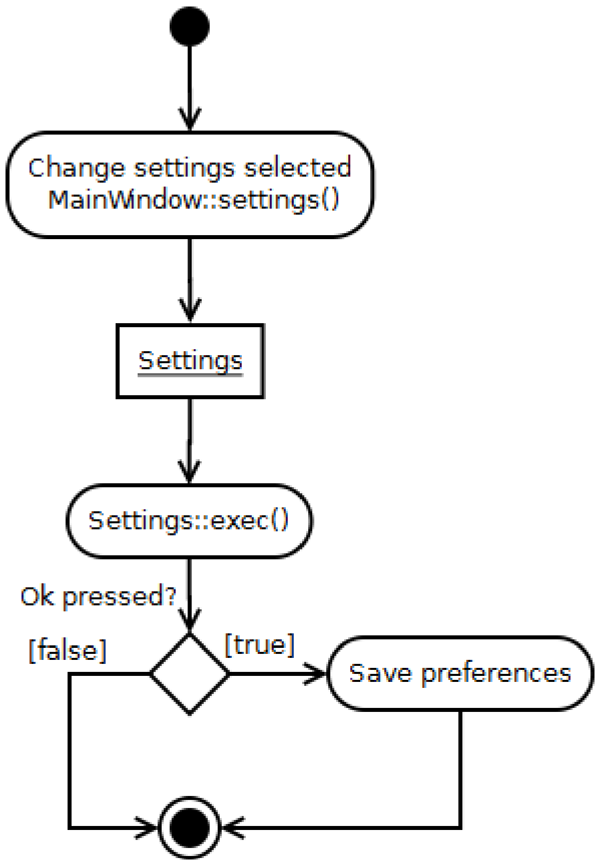


Fig. 27. Change Settings use case realization

Change Settings use case is given in the figure 27 and it handles the operation of changing application's settings.

The use case begins when the Radar Operator chooses the “change settings” action from the menu. This causes the settings method of the MainWindow to be called. A Setting object is created and its exec method called which subsequently displays a dialog for the Radar Operator.

The Radar Operator is able to change the application's settings using the dialog. When the ok button is pressed, the modified settings are written in the user preference's database which are subsequently read at the next application startup. The application therefore must be restarted in order for the setting changes to take any effect. If the cancel button is pressed then any changes done on the application's settings will not be saved but are ignored.

3.3.5. Use Case: Identify Ship

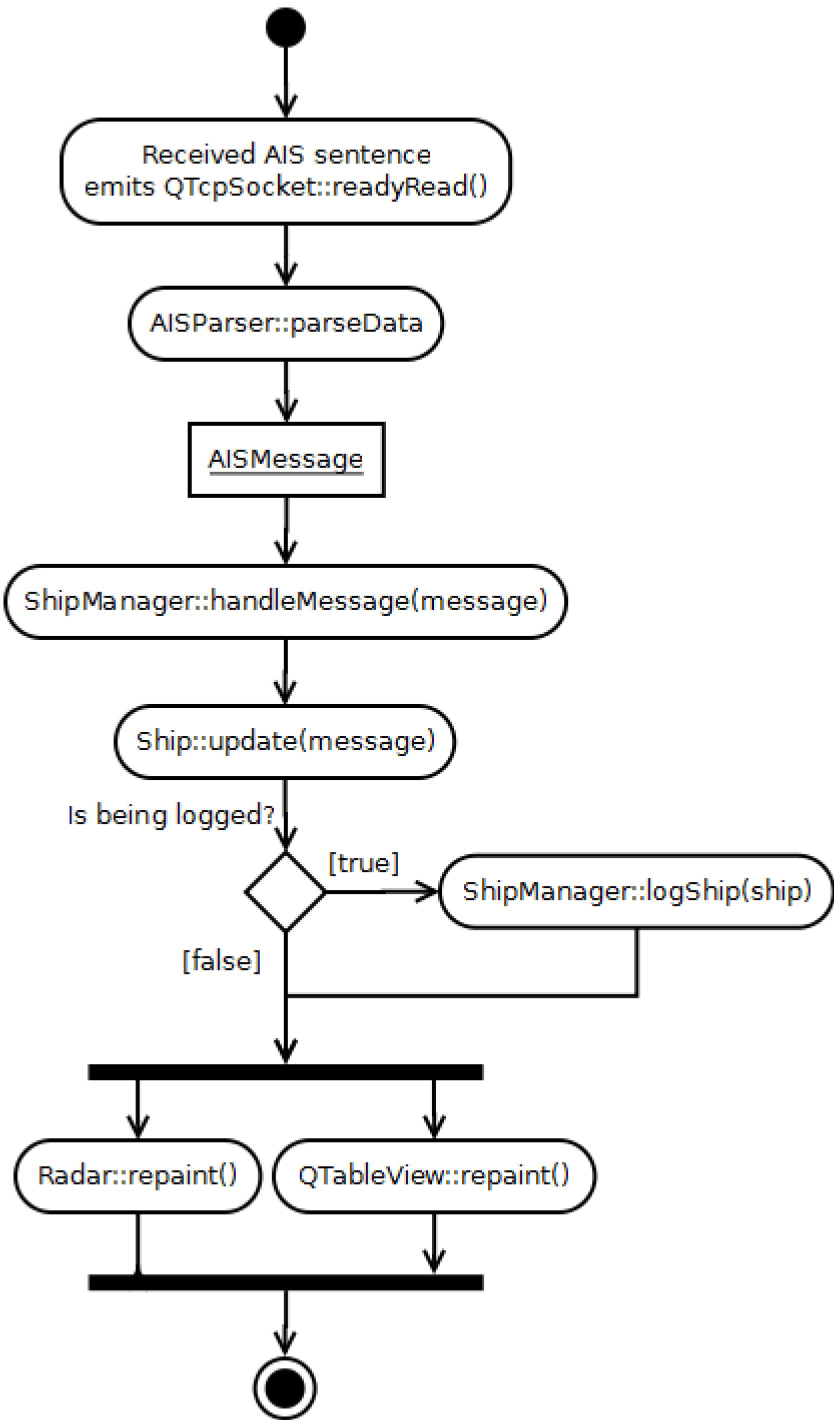


Fig. 28. Identify Ship use case realization

The figure 28 describes the Identify Ship use case. The Identify Ship use case handles the processing of the AIS protocol by parsing the raw AIVDM/AIVDO sentences into meaninful message objects. Each message object is used to update the ShipManager's internal cache of Ship objects which in turn are used to draw the list of ships and the radar map user interface.

The use case begins when the AIS send a sentence via a TCP/IP connection. The QTcpObject receives this data and as a response emits the readyRead signal. The signal is connected to the parseData slot of the AISParser object which handles the low-level parsing of the protocol. The following code listing illustrates this process:

```
void AISParser::parseData() {
     _sentenceBuffer.append(_socket->readAll());
    while (true) {
        int eol = _sentenceBuffer.indexOf("\n");
        if (eol == -1) break;
        parseSentence(_sentenceBuffer.left(eol));
        _sentenceBuffer.remove(0, eol + 1);
    }
}

void AISParser::parseSentence(const QByteArray &sentence) {
    if (sentence.startsWith("!AIVDM") || sentence.startsWith("!AIVDO")) {
        QList<QByteArray> fields = sentence.split(',');
        if (fields.count() < 7)
            return;
        uint fragmentCount = fields[1].toUInt();
        uint fragmentNumber = fields[2].toUInt();
        _payloadBuffer.append(fields[5]);
        if (fragmentNumber == fragmentCount) {
            parseMessage(_payloadBuffer, sentence.startsWith("!AIVDM"));
            _payloadBuffer.clear();
        }
    }
}

void AISParser::parseMessage(const QByteArray &message, bool aivdm) {
    Payload payload(message);
    uint type = payload.extractUInt(0, 6);
    if (type >= 1 && type <= 3) {
        PositionReport message(payload, aivdm);
        emit messageReceived(&message);
    } else if (type == 5) {
        ShipVoyageData message(payload, aivdm);
        emit messageReceived(&message);
    }
}
```

Since the received data is not necessary guaranteed to be send in a sentence-by-sentence basis (the operating system may arbitrarily cut the transmitted data into different sized packets) the parseData method temporiraly pushes all the incoming data into the so-called sentence buffer. From there the sentences are cut at newlines and each sentence is send for further processing by calling parseSentence method.

The parseSentence method processes each sentence further by splitting it into fields. In case of a multi-fragmented sentence the payload is pushed into the payload buffer which holds all the payloads accumulated so far for the currently pending message. When the end of the fragment is reached the payload is send for processing by passing it as an argument for the parseMessage method.

The parseMessage receives the payload and forwards it to the Payload class which subsequently decodes the armoring. A specific derived class of the AISMessage is constructed depending on the message's type. The AISMessage derived class decodes the data from the unarmored payload and stores everything into memory for later processing. Finally the messageReceived signal is emitted with the AISMessage object passed in as the argument to whatever further processing needed by the ShipManager.

The ShipManager receives the AISMessage object and updates its own internal state accordingly. What is done here depends on the type of the message received. For local position reports the ShipManager stores the local geographical coordinate of the AIS station. For a remote position report or a ship and voyage data report the ShipManager's internal cache of ship records are updated by calling update method of the Ship in question.

If the ship in question is tagged for logging then the ship's data is written to the end of a log file. Log files are named by the current data and are formatted in a line basis, each line beginning with a time stamp and followed by a semicolon separated list of data fields in a machine readable format.

The following is an example of a single line in a log file:

```
[2011-05-08T18:23:05]219000892;;;0;56.00247;11.32922;++;6.8;321.6;3.388;103.734;322.134
```

The data fields are the following:

- Time stamp enclosed in square brackets in the ISO 8601 standard format expressing the time when the event was logged.
- MMSI identifier.
- Call sign.
- Ship name.
- Navigation status code as described in the Appendix 3.
- Latitude expressed as degrees.
- Longitude expressed as degrees.
- Rate of turn expressed as degrees per minute or alternatively strings ++ or -- to indicate a rate of turn more than 5 degrees per 30 seconds to the right or left respectively.
- Speed over ground expressed as knots.
- Course over ground expressed as degrees.
- Distance from the AIS expressed as kilometers.
- Bearing angle expressed in degrees.
- Aspect angle expressed as degrees.

Any of the data fields except the time stamp and MMSI may be missing in which case the data was not available from the AIS.

Finally, both the ship list (as a QTableView object) and the Radar are repainted to reflect the changes in the user interface.

3.3.6. Use Case: Rotate Beam

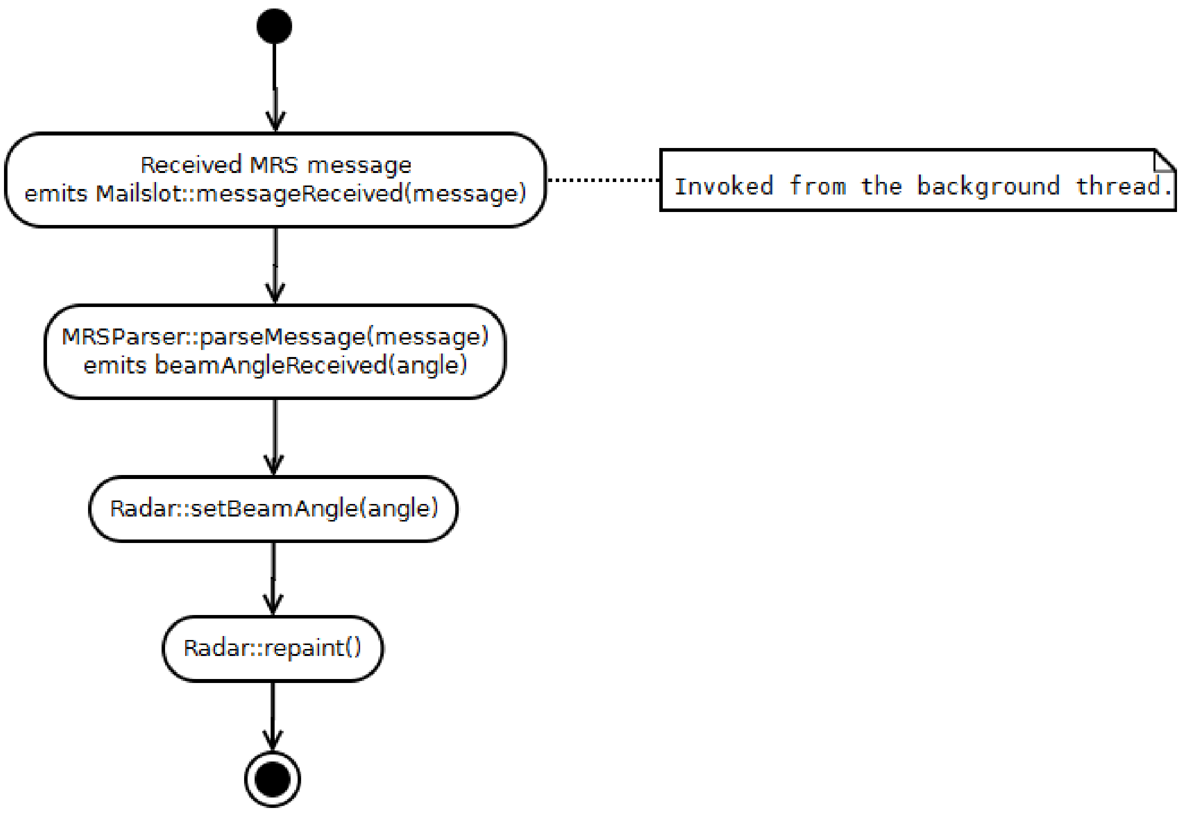

Received MRS message
emits Mailslot::messageReceived(message)
Invoked from the background thread.
MRSParser::parseMessage(message)
emits beamAngleReceived(angle)
Radar::setBeamAngle(angle)
Radar::repaint()

Fig. 29. Rotate Beam use case realization

The figure 29 describes the Rotate Beam use case that handles messages received from the MRS subsystem by extracting the radar beam angle from it and redrawing the superimposed radar beam on top of the radar map.

The use case begins when a Mailslot reads a message in the background thread which causes the messageReceived signal to get emitted.

The signal is processed in the parseMessage slot of the MRSParser. The slot extracts the radar beam angle from the message using regular expressions which is illustrated in the following code listing:

```
void MRSParser::parseMessage(const QString &message) {
    static const QRegExp pattern("[0-9]{3}[.,][0-9]{2};");
    if (pattern.indexIn(message) != -1) {
        qreal angle = parseAngle(pattern.cap());
        emit beamAngleReceived(angle);
    }
}
```

A note should be taken on the fact that the QRegExp is created on the stack using the static keyword – that is, the instance is created the first time the method is called and the same instance is reused on each subsequent method call. This is due to the fact that the regular expression given for the constructor must be compiled and processed into a form usable by the matching algorithm. By using the static keyword some CPU processing time is gained at the expense of memory since the regular expression will only be compiled just once.

After the radar beam angle is parsed from the message the MRSParser emits the beamAngleReceived slot which is connected to the setBeamAngle of the Radar instance. This causes the Radar object to update the beam angle and repaints the radar map.

4. CONCLUSIONS

The study conducted for the Danish Defence Acquistion and Logistics Organization involved designing and implementing a pointing aid application to be used with the Modular Radar System. The Automatic Identification System was employed as means for tracking and identifying vessels in the radar's vicinity in order to display them on a radar map.

Major part in conducting the study involved researching the interfaces used to communicate between the subsystems and getting the software to work with the existing control panel software. Since the actual radar unit was located on a remote military facility and therefore the time reserved for interacting with the radar was limited, additional testing software was required to be created which allowed the system to be emulated locally without actually being at the site itself.

The C++ programming language and the Qt framework proved to be very suitable design choices in implementing the application. Extensive class library of Qt provided the framework for building the application's user interface as well as communicating with the AIS using TCP/IP sockets. Since Qt did not understandably implement Window Mailslots IPC mechanism itself, a reusable object-oriented wrapper around the Windows Mailslots API was developed. Qt's signals and slots mechanism was able to handle communication between the application's components across thread boundaries by itself properly. All the services and functionality provided by the Qt framework greatly decreased the development time required to implement the Pointing and Tracking Aid application.

During the testing phase of the final application, two anomalies were observed.

First, one case was observed in which the AIS seemed to output completely garbled information for a ship. The ship list displayed the ship with completely invalid and nonsensical values such as zero MMSI and the rate of turn completely out of bounds. The only valid data fields were the call sign and ship name. Additionally the same ship (as identified by the call sign and ship name) was reported twice as another row in the ship list with all the data fields correctly displayed.

Secondly, ship distances relatively to the AIS contained significant inaccuracies. The magnitude of the error was observed to be directly proportional to the distance itself from the AIS. The inaccuracy for a ship of 4 kilometers away was in the range of 100 meters while for a ship of 20 kilometers away was in the magnitude of 1 kilometer. The actual distance of a ship was measured with the Modular Radar System itself.

The most probable reason for the first issue is either of the following two:

1. The message sent from the AIS to the application through the TCP/IP connection was for a reason or the other malformed somewhere between the connection line. The application interpreted the message as a genuine, containing information for a

ship with MMSI of zero. The ship manager created a record for the ship and the ship list displayed the data. Subsequent messages regarding the same ship were correctly formed with a correct MMSI, therefore creating an additional record in the ship manager while still retaining the previous malformed ship record.
2. The AIS sent a Ship and Voyage Data message for a ship before a Position Report message was sent for it and the application wasn't able to properly handle the case.

The solution for reason number one is to implement a check sum verification which is not currently implemented. Since sentences optionally contain a hexadecimal check sum at the end of each sentence, the check sum could be used to verify the validness of received sentences and to ignore ones which are found to be malformed. Also additional sanity checking could be implemented to verify the values received from the AIS.

The algorithm used for calculating the distance between the AIS and the ship contains many simplifications and assumptions, all of which might accumulate a small cumulative inaccuracy to the computed result. Perhaps the radius of the Earth used is not accurate enough. Perhaps the method of projecting the geographical latitude and longitude pair into a two-dimensional coordinate space and performing calculations using Cartesian coordinates is not suitable. Perhaps the Automatic Identification System reported its own geographical location inaccurately due to inadequate calibration or perhaps the Modular Radar System used to measure the distance was miscalibrated itself. There are many issues to consider and as of now the fundamental reason for the miscalculated distances remain unknown and more studies would be required. The fact that the magnitude of the error was proportional to the distance itself does give some suggestions to assume that the algorithm itself might not accurate enough.

There are also many other ways how the application could be extended both in terms of usability and technical features. The AIS parser could be improved to support the full range of AIVDM/AIVDO message types and the sentence check sum verification could be implemented for more reliability. The user interface could be improved by allowing column customization in the ship list for example by allowing sorting of ships by a chosen data field and allowing rearranging the columns by dragging. Much more data for each ship can also be collected from the AIS and displayed in the ship list. A tracking algorithm such as one employing a Kalman filter could be used to observe ships on the radar map in a real-time manner as opposed to the current pseudo real-time approach where each ship is updated every few seconds as new data is received from the AIS.

Another interesting project would be to extend the Pointing and Tracking Aid to work with air crafts using an IFF (Identification Friend or Foe) receiver. An IFF works in very similar fashion as an AIS except that it is being used for identifying air crafts instead of naval vessels.

On a more personal level, the project provided an interesting technical challenge and a rewarding experience in working with Danish scientists in the field of military applications. A hands-one experience with software engineering using the Qt framework was gained as well as in parsing binary-based protocol sentences.

REFERENCES

/1/ American National Standards Institute, ISO/IEC 9899:1999 Programming Languages - C.

/2/ GTK+ - About, [WWW-document], <http://www.gtk.org/> 9.5.2011.

/3/ Java Language Specification, Third Edition, [WWW-document], <http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html> 9.5.2011.

/4/ Kongsberg Seatex AS, Seatex AIS 200 P Instruction Manual, 2007-05-04.011.

/5/ Microsoft Corporation, About Mailslots (Windows), [WWW-document], <http://msdn.microsoft.com/en-us/library/aa365130%28v=vs.85%29.aspx> 21.4.2011.

/6/ Microsoft Corporation, Mailslot Names (Windows), [WWW-document], <http://msdn.microsoft.com/en-us/library/aa365581%28v=vs.85%29.aspx> 21.4.2011.

/7/ Microsoft Corporation, MFC Reference, [WWW-document], <http://msdn.microsoft.com/en-us/library/d06h2x6e%28v=VS.100%29.aspx> 9.5.2011.

/8/ Møller Jesper, M.Sc. EE, Initial project meeting, 28.1.2011.

/9/ Møller Jesper, M.Sc. EE, Project description, 31.1.2011.

/10/ National Marine Electronics Association, NMEA, [WWW-document], <http://www.nmea.org/content/nmea_standards/nmea_083_v_400.asp> 9.5.2011.

/11/ Nokia Corporation, Qt – Cross-platform application and UI framework, [WWW-document], <http://qt.nokia.com/products> 21.4.2011.

/12/ Nokia Corporation, Qt in Use, [WWW-document], <http://qt.nokia.com/qt-in-use> 9.5.2011.

/13/ Oracle, Java Native Interface Specification, [WWW-document], <http://gpsd.berlios.de/AIVDM.html> 21.4.2011.

/14/ Raymond Eric S., AIVDM/AIVDO protocol decoding, [WWW-document], <http://download.oracle.com/javase/6/docs/technotes/guides/jni/spec/jniTOC.html> 30.4.2011.

/15/ Who Uses wxWidgets? - wxWidgets, [WWW-document], <http://www.wxwidgets.org/about/whouses.htm> 9.5.2011.

LIST OF APPENDICES

APPENDIX 1: PAYLOAD ARMORING

The information in this appendix is gathered from Eric S. Raymond's article AIVDM/AIVDO protocol encoding which can be found online from:

http://gpsd.berlios.de/AIVDM.html

AIS messages are encoded into the NMEA 0183 protocol sentences using a special six-bit payload armoring. Each ASCII character in the payload expresses a six-bit nibble. To derive the nibble value from a single payload character, subtract 48 from it and if the result is higher than 40, subtract 8. Concatenate all nibble with the most significant bit first to get the binary presentation of the whole message. The table 2 gives the corresponding decoded nibble for each payload character.

Table 2. Payload armoring

Payload Character	ASCII Value	Decoded Value	Binary Nibble
0	48	0	000000
1	49	1	000001
2	50	2	000010
3	51	3	000011
4	52	4	000100
5	53	5	000101
6	54	6	000110
7	55	7	000111
8	56	8	001000
9	57	9	001001
:	58	10	001010
;	59	11	001011
<	60	12	001100
=	61	13	001101
>	62	14	001110
?	63	15	001111
@	64	16	010000
A	65	17	010001
B	66	18	010010
C	67	19	010011
D	68	20	010100
E	69	21	010101
F	70	22	010110

Payload Character	ASCII Value	Decoded Value	Binary Nibble
G	71	23	010111
H	72	24	011000
I	73	25	011001
J	74	26	011010
K	75	27	011011
L	76	28	011100
M	77	29	011101
N	78	30	011110
O	79	31	011111
P	80	32	100000
Q	81	33	100001
R	82	34	100010
S	83	35	100011
T	84	36	100100
U	85	37	100101
V	86	38	100110
W	87	39	100111
`	96	40	101000
a	97	41	101001
b	98	42	101010
c	99	43	101011
d	100	44	101100
e	101	45	101101
f	102	46	101110
g	103	47	101111
h	104	48	110000
i	105	49	110001
j	106	50	110010
k	107	51	110011
l	108	52	110100
m	109	53	110101
n	110	54	110110
o	111	55	110111
p	112	56	111000
q	113	57	111001
r	114	58	111010

Payload Character	ASCII Value	Decoded Value	Binary Nibble
s	115	59	111011
t	116	60	111100
u	117	61	111101
v	118	62	111110
w	119	63	111111

APPENDIX 2: PAYLOAD CHARACTER ENCODING

The information in this appendix is gathered from Eric S. Raymond's article AIVDM/AIVDO protocol encoding which can be found online from:

http://gpsd.berlios.de/AIVDM.html

The AIS protocol uses a special character encoding to express text. Each character is expressed using a six-bit nibble and is mapped into an ASCII character. Character strings are terminating with the @ character which should not be considered as being part of the decoded text. The table 3 gives the corresponding ASCII character for each six-bit binary nibble.

Table 3. Character encoding

Binary Nibble	Decimal Value	ASCII Character
000000	0	@
000001	1	A
000010	2	B
000011	3	C
000100	4	D
000101	5	E
000110	6	F
000111	7	G
001000	8	H
001001	9	I
001010	10	J
001011	11	K
001100	12	L
001101	13	M
001110	14	N
001111	15	O
010000	16	P
010001	17	Q
010010	18	R
010011	19	S
010100	20	T
010101	21	U
010110	22	V
010111	23	W

Binary Nibble	**Decimal Value**	**ASCII Character**
011000	24	X
011001	25	Y
011010	26	Z
011011	27	[
011100	28	\
011101	29	]
011110	30	^
011111	31	_
100000	32	(space)
100001	33	!
100010	34	“
100011	35	#
100100	36	$
100101	37	%
100110	38	&
100111	39	'
101000	40	(
101001	41	)
101010	42	*
101011	43	+
101100	44	,
101101	45	-
101110	46	.
101111	47	/
110000	48	0
110001	49	1
110010	50	2
110011	51	3
110100	52	4
110101	53	5
110110	54	6
110111	55	7
111000	56	8
111001	57	9
111010	58	:
111011	59	;

Binary Nibble	**Decimal Value**	**ASCII Character**
111100	60	<
111101	61	=
111110	62	>
111111	63	?

APPENDIX 3: PAYLOAD MESSAGES

The information in this appendix is gathered from Eric S. Raymond's article AIVDM/AIVDO protocol encoding which can be found online from:

http://gpsd.berlios.de/AIVDM.html

Each payload message begins with a 38-bit header which is described in detail in the table 3.

Table 4. AIS message header

Bits	Length	Datatype	Description
0-5	6	Unsigned integer	Message type.
6-7	2	Unsigned integer	Repeat indicator.
8-37	30	Unsigned integer	A 9-digit Maritime Mobile Service Identifier (MMSI).

The repeat indicator field is a directive for an AIS transceivers to rebroadcast the same message forward and increase the indicator by one on each rebroadcast. A value of three indicates ”do not repeat”. This is intended to implement a rudimentary form of routing to get around obstructions such as hills.

The MMSI field is a unique 9-digit identifier for the ship's radio. First three digits indicate the country in which the identifier was issued. Denmark uses country codes 219 and 220.

The message type field specifies the structure of the message which follows after the header. The message types are listed in the table 5.

Table 5. AIS message types

Message Type	Message Class
1	Position Report Class A
2	Position Report Class A (Assigned schedule)
3	Position Report Class A (Response to interrogation)
4	Base Station Report
5	Ship and Voyage Data
6	Addressed Binary Message
7	Binary Acknowledge
8	Binary Broadcast Message
9	Standard SAR Aircraft Position Report
10	UTC and Date Inquiry
11	UTC and Date Response
12	Addressed Safety Related Message

Message Type	Message Class
13	Safety Related Acknowledge
14	Safety Related Broadcast Message
15	Interrogation
16	Assigned Mode Command
17	GNSS Binary Broadcast Message
18	Standard Class B CS Position Report
19	Extended Class B Equipment Position Report
20	Data Link Management
21	Aid-to-Navigation Report
22	Channel Management
23	Group Assignment Command
24	Class B CS Static Data Report
25	Binary Message, Single Slot
26	Binary Message, Multiple Slot
27	Position Report for Long-Range Applications

Message classes Position Report Class A and Ship and Voyage Data are explained below. Other message types are omitted here since they are not relevant to the Pointing and Tracking Aid application.

Position Report Class A

Under normal operations an AIS transceiver mounted on a vessel will broascast a position report every 2 to 10 seconds depending on the vessel's speed while underway and every 3 minutes while anchored and stationary. The table 6 gives the structure of the Position Report Class A message.

Table 6. Position Report Class A message type

Bits	Length	Datatype	Description
38-41	4	Unsigned integer	Navigation status.
42-49	8	Signed integer	Rate of turn.
50-59	10	Unsigned integer	Speed over ground.
60-60	1	Boolean flag	Position accuracy.
61-88	28	Signed integer	Longitude expressed as minutes/10000.
89-115	27	Signed integer	Latitude expressed as minutes/10000.
116-127	12	Unsigned integer	Course over ground relative to North.
128-136	9	Unsigned integer	True heading relative to North.
137-142	6	Unsigned integer	Second of an UTC time stamp.

Bits	Length	Datatype	Description
143-144	2	Unsigned integer	Maneuver indicator.
145-147	2		Not used.
148-148	1	Boolean flag	RAIM flag.
149-167	19	Unsigned integer	Radio status.

The table 7 describes the values of the navigation status field.

Table 7. Navigation status

Navigation Status	Description
0	Under way using engine.
1	At anchor.
2	Not under command.
3	Restricted manoeuvrability.
4	Constrained by her draught.
5	Moored.
6	Aground.
7	Engaged in fishing.
8	Under way sailing.
9	Reserved for future use.
10	Reserved for future use.
11	Reserved for future use.
12	Reserved for future use.
13	Reserved for future use.
14	Reserved for future use.
15	Not defined or default.

The rate of turn field has the following meanings:

- Value of 0 indicates the vessel is not turning.
- Values between 1 and 126 indicates the vessel is currently turning right.
- Values between -1 and -126 indicates the vessel is currently turning left.
- Value of 127 indicates the vessel is turning right at more than 5 degrees per 30 seconds.
- Value of -127 indicates the vessel is turning left at more than 5 degrees per 30 seconds.
- Value of 128 indicates no turn information is available.

The actual rate of turn is calculated using the following Mathematical formula:

$$\left(\frac{\textit{Rate of Turn}}{4.733}\right)^2 \text{degrees per minute}$$

The speed over ground field indicates the speed of the vessel in knots expressed in 0.1 precision from 0 to 102 knots. For example the value of 101 means the vessel is proceeding in 10.1 knots velocity. A special value 1023 indicates speed information is not available.

The longitude and latitude fields give the geographic location of the vessel expressed as 1/10000th of a minute accuracy. To obtain the coordinate as a degree divide the value by 600000. A positive value of longitude increases to East and a positive value of latitude increases to North. A special longitude value of 181 degrees indicates the longitude is not available or is the default while a latitude value of 91 degrees indicates the same.

The course over ground indicates the course of the vessel in degrees expressed with a 0.1 precision relatively to North. A value of 1205 therefore indicates a course of 120.5 degrees. A special value of 3600 indicates no course information is available.

Rest of the fields are not used by the Pointing and Tracking Aid application so therefore they are omitted here for clarity.

Ship and Voyage Data

Ship and Voyage Data will be broadcasted every 6 minutes by sailing vessels. The structure of this message type is given in the table 8.

Table 8. Ship and Voyage Data message type

Bits	Length	Datatype	Description
38-39	2	Unsigned integer	AIS version.
40-69	30	Unsigned integer	IMO number.
70-111	42	String	Call sign.
112-231	120	String	Vessel name.
232-239	8	Unsigned integer	Ship type.
240-248	9	Unsigned integer	Dimension to bow in meters.
249-257	9	Unsigned integer	Dimension to stern in meters.
258-263	6	Unsigned integer	Dimension to port in meters.
264-269	6	Unsigned integer	Dimension to starboard in meters.
270-273	4	Unsigned integer	Position fix type.
274-277	4	Unsigned integer	Estimated time of arrival: month.
278-282	5	Unsigned integer	Estimated time of arrival: day.
283-287	5	Unsigned integer	Estimated time of arrival: hour.

Bits	Length	Datatype	Description
288-293	6	Unsigned integer	Estimated time of arrival: minute.
294-301	8	Unsigned integer	Draught expressed as meters/10.
302-421	120	String	Destination.
422-422	1	Boolean flag	DTE.
423-423	1		Not used.

The only fields from this message class which are used in the Pointing and Tracking Aid applications are the call sign and ship name fields. These fields are encoded as character strings using the special AIS text encoding as explained in Appendix 2.

APPENDIX 4: USE CASES

Use case: Tag Ship
Brief description: The Radar Operator tags a ship for logging. Subsequent messages received from the AIS regarding the ship causes the ship's data to be logged into a log file.
Primary actors: Radar Operator
Secondary actors: None
Preconditions: None
Main flow: 1 The use case begins when the Radar Operator tags a ship. 2 The application set the logging flag on for the ship.
Postconditions: 1 The ship was flagged on for logging.
Alternative flows: None

Use case: Untag Ship
Brief description: The Radar Operator untags a ship from being logged. Subsequent messages received from the AIS regarding the ship no longer causes the ship's data to be logged.
Primary actors: Radar Operator
Secondary actors: None
Preconditions: None
Main flow: 1 The use case begins when the Radar Operator untags a ship. 2 The application sets the logging flag off for the ship.
Postconditions: 1 The ship was flagged off from being logged.
Alternative flows: None

Use case: Select Ship
Brief description: The Radar Operator selects a ship from the application's user interface. The application highlights a ship to give visual feedback of the selection.
Primary actors: Radar Operator
Secondary actors: None
Preconditions: None
Main flow: 1 The use case begins when the Radar Operator selects a ship. 2 The application updates its internal state of the currently selected ship. 3 If a new ship was selected 3.1 The application highlights the ship on the radar map and the ship list.
Postconditions: 1 The application highlighted the selected ship if one was selected.
Alternative flows: None

Use case: Change Settings
Brief description: The Radar Operator changes application's settings.
Primary actors: Radar Operator
Secondary actors: None
Preconditions: None
Main flow: 1 The use case begins when the Radar Operator selects “change settings” action. 2 The application displays a settings dialog for the Radar Operator. 3 The Radar Operator changes settings. 4 If the ok button is pressed 4.1 The application stores the settings into a user preferences database.
Postconditions: None
Alternative flows: None

Use case: Identify Ship

Brief description:
The AIS sends a message regarding a remote naval vessel or the local station itself.

Primary actors:
AIS

Secondary actors:
None

Preconditions:
None

Main flow:

```
1 The use case begins when the AIS sends a message.
2 The application decodes the message's payload armoring and finds the message
  type and the MMSI identification number.
3 The application decodes the rest of the data found in the payload according to the
  message's type.
4 If the message describes a remote naval vessel
    4.1 If a ship record with the specified MMSI already exists
        4.1.1 The application updates the existing ship record from the decoded data.
    4.2 Else
        4.2.1 The application creates a new ship record using the decoded data.
5 Else
    5.1 The application updates the local station's record.
```

Postconditions:

1 The application handled the message.

Alternative flows:
None

Use case: Rotate Beam
Brief description: The MRS sends a message regarding changes in the current radar beam angle. The application decodes the message and updates the radar map to reflect changes.
Primary actors: MRS
Secondary actors: None
Preconditions: None
Main flow: 1 The use case begins when the MRS sends a message. 2 The application decodes the radar beam angle from the message. 3 The application redraws the radar beam angle on the radar map.
Postconditions: 1 The application handled the message. 2 The application updated the radar map.
Alternative flows: Non

APPENDIX 5: SOURCE CODE

List of source files

- aismessages.cpp: AIS Message classes.
- aismessage.h: AIS Message classes.
- aisparser.cpp: Automatic Identification System protocol parser.
- aisparser.h: Automatic Identification System protocol parser.
- constants.cpp: Compile-time constants.
- constants.h: Compile-time constants.
- coord.cpp: Geographic coordinate class.
- coord.h: Geographic coordinate class.
- dataformatter.cpp: AIS Data Formatter class.
- dataformatter.h: AIS Data Formatter class.
- mailslot.cpp: Object-oriented Windows Mailslots API wrapper.
- mailslot.h: Object-oriented Windows Mailslots API wrapper.
- main.cpp: Application main entry-point.
- mainwindow.cpp: Main Window class.
- mainwindow.h: Main Window class.
- mrsparser.cpp: Modular Radar System protocol parser.
- mrsparser.h: Modular Radar System protocol parser.
- payload.cpp: AIS Message Payload decoder.
- payoad.h: AIS Message Payload decoder.
- radar.cpp: Radar user interface component.
- radar.h: Radar user interface component.
- settings.cpp: Settings dialog.
- settings.h: Settings dialog.
- ship.cpp: Ship data encapsulation.
- ship.h: Ship data encapsulation.
- shipmanager.cpp: Ship Manager class.
- shipmanager.h: Ship Manager class.

File: aismessage.cpp

```
/*
 * Pointing and Tracking Aid
 *
 * This software was written by Henri Häkkinen as part of his Bachelor's thesis for
 * the Danish Defence Acquistion and Logistics Organization during Spring 2011.
 *
 * aismessages.cpp
 *
 * Implementation of the AIVDM/AIVDO protocol messages classes.
 */

#include "aismessages.h"
#include "payload.h"
```

```
AISMessage::AISMessage(const Payload &payload, bool aivdm) {
    _aivdm = aivdm;
    _type = payload.extractUInt(0, 6);
    _mmsi = payload.extractUInt(8, 30);
}

PositionReport::PositionReport(const Payload &payload, bool aivdm) :
    AISMessage(payload, aivdm)
{
    _status = payload.extractUInt(38, 4);
    _turn = payload.extractInt(42, 8);
    _speed = payload.extractUInt(50, 10);
    _lon = payload.extractInt(61, 28);
    _lat = payload.extractInt(89, 27);
    _course = payload.extractUInt(116, 12);
}

ShipVoyageData::ShipVoyageData(const Payload &payload, bool aivdm) :
    AISMessage(payload, aivdm)
{
    _callSign = payload.extractString(70, 42);
    _shipName = payload.extractString(112, 120);
}
```

File: aismessage.h

```
/*
 * Pointing and Tracking Aid
 *
 * This software was written by Henri Häkkinen as part of his Bachelor's thesis for
 * the Danish Defence Acquistion and Logistics Organization during Spring 2011.
 *
 * aismessages.h
 *
 * AIVDM/AIVDO protocol message classes.
 *
 * Protocl reference:
 *  - http://gpsd.berlios.de/AIVDM.html
 */

#ifndef AISMESSAGES_H
#define AISMESSAGES_H

#include <QObject>
#include "coord.h"

class Payload;

class AISMessage : public QObject {
    Q_OBJECT

public:
    AISMessage(const Payload &payload, bool aivdm);
    virtual ~AISMessage() {}

    bool    isAIVDM() const     { return _aivdm; }
    uint    type() const        { return _type; }
    uint    mmsi() const        { return _mmsi; }

private:
    bool    _aivdm;             // AIVDM or AIVDO sentence?
    uint    _type;              // Message Type
    uint    _mmsi;              // Mobile Marine Service Identifier
};

// Type 1, 2 and 3: Position Report Class A
class PositionReport : public AISMessage {
    Q_OBJECT

public:
    PositionReport(const Payload &payload, bool aivdm);

    uint    status() const      { return _status; }
    int     turn() const        { return _turn; }
    uint    speed() const       { return _speed; }
    Coord   coord() const       { return Coord(_lat, _lon); }
    int     lon() const         { return _lon; }
    int     lat() const         { return _lat; }
    uint    course() const      { return _course; }

private:
    uint    _status;            // Navigation Report
    int     _turn;              // Rate of Turn
    uint    _speed;             // Speed Over Ground
    int     _lon;               // Longitude (Minutes/10000)
    int     _lat;               // Latitude (Minutes/10000)
    uint    _course;            // Course Over Ground
};

// Type 5: Ship and Voyage Data
class ShipVoyageData : public AISMessage {
    Q_OBJECT

public:
    ShipVoyageData(const Payload &payload, bool aivdm);

    QString callSign() const    { return _callSign; }
    QString shipName() const    { return _shipName; }

private:
    QString _callSign;          // Call Sign
    QString _shipName;          // Vessel Name
};

#endif // AISMESSAGES_H
```

File: aisparser.cpp

```
/*
 * Pointing and Tracking Aid
 *
 * This software was written by Henri Häkkinen as part of his Bachelor's thesis for
 * the Danish Defence Acquistion and Logistics Organization during Spring 2011.
 *
 * aisparser.cpp
 *
 * Implementation of the AISParser class.
 */

#include <QMessageBox>
#include <QTcpSocket>
#include "aismessages.h"
#include "aisparser.h"
#include "payload.h"
```

```
AISParser::AISParser(QTcpSocket *socket, QObject *parent) :
    QObject(parent),
    _socket(socket)
{
    connect(_socket, SIGNAL(readyRead()), this, SLOT(parseData()));
    connect(_socket, SIGNAL(error(QAbstractSocket::SocketError)), this, SLOT(error()));
}

void AISParser::error() {
    // Called when an error occurs on the TCP/IP connection.
    QMessageBox::critical(0, "Error", "AIS connection error:\n" + _socket->errorString());
}

void AISParser::parseData() {
    // Called when there is an incoming data coming from the AIS transceiver.

    _sentenceBuffer.append(_socket->readAll());
    while (true) {
        int eol = _sentenceBuffer.indexOf("\n");
        if (eol == -1) break;
        parseSentence(_sentenceBuffer.left(eol));
        _sentenceBuffer.remove(0, eol + 1);
    }
}

void AISParser::parseSentence(const QByteArray &sentence) {
    // Parse an AIVDM/AIVDO protocol sentence.  Sentences not beginning with !AIVDM or
    // !AIVDO or otherwise malformatted sentences are silently ignored.
    //
    // Note: sentence checksum verification is not implemented!

    if (sentence.startsWith("!AIVDM") || sentence.startsWith("!AIVDO")) {
        QList<QByteArray> fields = sentence.split(',');
        if (fields.count() < 7)
            return;

        uint fragmentCount = fields[1].toUInt();
        uint fragmentNumber = fields[2].toUInt();
        _payloadBuffer.append(fields[5]);

        if (fragmentNumber == fragmentCount) {
            parseMessage(_payloadBuffer, sentence.startsWith("!AIVDM"));
            _payloadBuffer.clear();
        }
    }
}

void AISParser::parseMessage(const QByteArray &message, bool aivdm) {
    // Parse a message contained in the armored payload.

    Payload payload(message);
    uint type = payload.extractUInt(0, 6);

    if (type >= 1 && type <= 3) {
        PositionReport message(payload, aivdm);
        emit messageReceived(&message);
    } else if (type == 5) {
        ShipVoyageData message(payload, aivdm);
        emit messageReceived(&message);
    }
}
```

File: aisparser.h

```
/*
 * Pointing and Tracking Aid
 *
 * This software was written by Henri Häkkinen as part of his Bachelor's thesis for
 * the Danish Defence Acquistion and Logistics Organization during Spring 2011.
 *
 * aisparser.h
 *
 * Automatic Identification System (AIS) message parser.  Parses messages received over
 * TCP/IP network into high-level objects and emits signals as things are recognized from
 * the input stream.
 *
 * Protocol reference:
 * - http://gpsd.berlios.de/AIVDM.html
 */

#ifndef AISPARSER_H
#define AISPARSER_H

#include <QObject>

class QTcpSocket;
class AISMessage;

class AISParser : public QObject {
    Q_OBJECT

public:
    AISParser(QTcpSocket *socket, QObject *parent = 0);

signals:
    void messageReceived(const AISMessage *message);

private slots:
    void error();
    void parseData();

private:
    void parseSentence(const QByteArray &sentence);
    void parseMessage(const QByteArray &payload, bool aivdm);

    QTcpSocket *_socket;
    QByteArray _sentenceBuffer;
    QByteArray _payloadBuffer;
};

#endif // AISPARSER_H
```

File: constants.cpp

```
/*
 * Pointing and Tracking Aid
 *
 * This software was written by Henri Häkkinen as part of his Bachelor's thesis for
 * the Danish Defence Acquistion and Logistics Organization during Spring 2011.
 *
 * constants.cpp
 *
 * Compile-time constants.
 */

#include <cmath>
#include "constants.h"

const QString OrganizationName      = "Danish Defence Acquistion and Logistics 
Organization";
const QString OrganizationDomain    = "forsvaret.dk";
const QString ApplicationName       = "Pointing and Tracking Aid";
const QString ApplicationVersion    = "1.0";
const QString ApplicationIconName   = "icon48.ico";
const QString LogDirectory          = "logs";
const QString LogFileExtension      = ".txt";
const QString MRSMailslotName       = "MRS-Mail";
const QString AISHostname           = "192.168.127.254";
const quint16 AISPort               = 4001;
const qreal   RadarDefaultLatitude  = 56.0097;
const qreal   RadarDefaultLongitude = 11.2784;
const qreal   RadarDefaultZoom      = 0.01;
const qreal   RadarBeamAngle        = 3.5 * M_PI/180.0;
const qreal   ShipSize              = 2.0;
const QColor  ShipColorRegular      = QColor(255, 0, 0);
const QColor  ShipColorSelected     = QColor(0, 0, 255);
const qreal   EarthRadius           = 6371.0;
```

File: constants.h

```
/*
 * Pointing and Tracking Aid
 *
 * This software was written by Henri Häkkinen as part of his Bachelor's thesis for
 * the Danish Defence Acquistion and Logistics Organization during Spring 2011.
 *
 * constants.h
 *
 * Compile-time constants.
 */

#ifndef CONSTANTS_H
#define CONSTANTS_H

#include <QtGlobal>
#include <QString>
#include <QColor>
#include "coord.h"

// Name of the organization.
extern const QString OrganizationName;

// Domain name of the organzation.
extern const QString OrganizationDomain;

// Full application name.
extern const QString ApplicationName;

// Application version string.
extern const QString ApplicationVersion;

// Application icon filename.
extern const QString ApplicationIconName;

// Directory where ship logs are being stored.
// Don't include a trailing path separator!
extern const QString LogDirectory;

// File extension of a log file.
extern const QString LogFileExtension;

// Name of the MRS mailslot to listen to.
extern const QString MRSMailslotName;

// Hostname / IP address of the AIS server.
extern const QString AISHostname;

// Port number of the AIS server.
extern const quint16 AISPort;

// Default radar origin.
extern const qreal RadarDefaultLatitude;
extern const qreal RadarDefaultLongitude;

// Default radar zoom.
extern const qreal RadarDefaultZoom;

// Radar beam cut-off angle in radians.
extern const qreal RadarBeamAngle;

// Size of a ship drawn on the radar.
extern const qreal ShipSize;

// Color of a ship drawn on the radar.
extern const QColor ShipColorRegular;
extern const QColor ShipColorSelected;

// Earth's radius in kilometers.
extern const qreal EarthRadius;

#endif // CONSTANTS_H
```

File: coord.cpp

```
/*
 * Pointing and Tracking Aid
 *
 * This software was written by Henri Häkkinen as part of his Bachelor's thesis for
 * the Danish Defence Acquistion and Logistics Organization during Spring 2011.
 *
 * coord.cpp
 *
 * Implementation of the Coord class.
 */

#include <cmath>
#include "constants.h"
#include "coord.h"

const int Coord::UndefinedLatitude = 91 * 600000;
const int Coord::UndefinedLongitude = 181 * 600000;
const Coord Coord::Undefined(91 * 600000, 181 * 600000);

qreal Coord::distance(const Coord &ca, const Coord &cb) {
    // Calculates the distance from point 'ca' to point 'cb' as kilometers.
    // Returns -1 if either of the points is 'undefined'.

    if (ca.isUndefined() || cb.isUndefined())
        return -1;

    const qreal dx = cb.x() - ca.x();
    const qreal dy = cb.y() - ca.y();
    return EarthRadius * sqrt(dx*dx + dy*dy);
}

qreal Coord::bearing(const Coord &ca, const Coord &cb) {
    // Calculates the bearing from point 'ca' to point 'cb' relatively to the reference
    // angle which points to North.  The returned angle is expressed in degrees and is
    // always normalized into the range [0, 360]. A -1 is returned if either of the passed
    // coordinates is 'undefined'.
    //
    // Note that the order which the two input coordinates are passed to this function
    // is significant!  bearing(a, b) is different from bearing(b, a)!

    if (ca.isUndefined() || cb.isUndefined())
        return -1;

    const qreal dx = cb.x()- ca.x();
    const qreal dy = cb.y()- ca.y();
    const qreal brn = atan2(dx, dy) * 180.0 / M_PI;
    return fmod(brn + 360.0, 360.0);
}

qreal Coord::aspect(const Coord &ca, const Coord &cb, uint course) {
    // Calculates the aspect from point 'ca' to point 'cb' with 'course' defining the
    // course over ground of a ship located at point 'cb'.  The 'course' is the value
    // received from the AIS and is expressed with 0.1 degree precision -- a value of
    // 105 means 10.5 degrees and the value therefore must be divided by 10 to get the
    // angle in degrees.
    //
    // A -1 is returned if either of the points are 'undefined'.  Again, the order which
    // the two locations are passed to this function is significant!

    if (ca.isUndefined() || cb.isUndefined())
        return -1;

    const qreal aspect = bearing(cb, ca) - course/10.0;
    return fmod(aspect + 360.0, 360.0);
}

Coord::Coord() :
    _lat(UndefinedLatitude),
    _lon(UndefinedLongitude)
{ /* nothing to do here */ }

Coord::Coord(int lat, int lon) :
    _lat(lat),
    _lon(lon)
{
    // Project the geographical location into a 2D Cartesian plane.
    qreal latRad = latRadians();
    qreal lonRad = lonRadians();
    _x = lonRad * cos(latRad);
    _y = latRad;
}
```

File: coord.h

```
/*
 * Pointing and Tracking Aid
 *
 * This software was written by Henri Häkkinen as part of his Bachelor's thesis for
 * the Danish Defence Acquistion and Logistics Organization during Spring 2011.
 *
 * coord.h
 *
 * Geographic Coordinate datatype class.
 *
 * A coordinate may be 'undefined' which does not express any real geographical
 * location.  The 'isUndefined' returns true for these kinds of coordinates.
 */

#ifndef COORD_H
#define COORD_H

#include <QtGlobal>

class Coord {
public:
    static const int UndefinedLatitude;
    static const int UndefinedLongitude;
    static const Coord Undefined;

    static qreal distance(const Coord &ca, const Coord &cb);
    static qreal bearing(const Coord &ca, const Coord &cb);
    static qreal aspect(const Coord &ca, const Coord &cb, uint course);

    Coord();
    Coord(int lat, int lon);

    int lat() const { return _lat; } // Minutes/10000
    int lon() const { return _lon; } // Minutes/10000
    qreal latDegrees() const { return _lat / 600000.0; }
    qreal lonDegrees() const { return _lon / 600000.0; }
    qreal latRadians() const { return latDegrees() * 0.0174532925; }
    qreal lonRadians() const { return lonDegrees() * 0.0174532925; }
    qreal x() const { return _x; }
    qreal y() const { return _y; }

    bool isUndefined() const {
        return (_lat == UndefinedLatitude) || (_lon == UndefinedLongitude);
    }

private:
    int _lat, _lon;     // latitude & longitude expressed as 1/10000th of a minute
    qreal _x, _y;       // projected 2D Cartesian coordinate
};

#endif // COORD_H
```

File: dataformatter.cpp

```
/*
 * Pointing and Tracking Aid
 *
 * This software was written by Henri Häkkinen as part of his Bachelor's thesis for
 * the Danish Defence Acquistion and Logistics Organization during Spring 2011.
 *
 * dataformatter.cpp
 *
 * Implementation of the DataFormatter class.
 */

#include <cmath>
#include "dataformatter.h"
#include "coord.h"

// Unicode Character 'Degree sign' (U+00B0).
static const QChar UnicodeDegreeSymbol(0x00B0);

QString DataFormatter::mmsi(uint mmsi) {
    // Formats the numeric MMSI as a text string.
    return QString::number(mmsi);
}

QString DataFormatter::callSign(const QString &callSign) {
    // Returns the call sign as is unless it's empty.
    if (callSign.isEmpty()) {
        return QString("Not defined");
    }
    return callSign;
}

QString DataFormatter::shipName(const QString &shipName) {
    // Returns the ship name as is unless it's empty.
    if (shipName.isEmpty()) {
        return QString("Not defined");
    }
    return shipName;
}

QString DataFormatter::status(uint status) {
    // Formats the navigation status as a human-readable text.

    switch (status) {
    case 0:     return QString("Under way using engine");
    case 1:     return QString("At anchor");
    case 2:     return QString("Not under command");
    case 3:     return QString("Restricted manoeuverability");
    case 4:     return QString("Constrained by her draught");
    case 5:     return QString("Moored");
    case 6:     return QString("Aground");
    case 7:     return QString("Engaged in fishing");
    case 8:     return QString("Under way sailing");

    // Status codes 9-14 are "Reserved for future amendment of Navigation Status for HSC"
    // according to the unofficial protocol specification.  We will just return "Not
    // defined" for them here.

//    case 9:     return QString("Reserved for future use");
//    case 10:    return QString("Reserved for future use");
//    case 11:    return QString("Reserved for future use");
//    case 12:    return QString("Reserved for future use");
//    case 13:    return QString("Reserved for future use");
//    case 14:    return QString("Reserved for future use");

    }

    return QString("Not defined");
}
```

```
QString DataFormatter::turn(int turn) {
    // Formats the rate of turn as a human-readable text.

    if (turn == 0) {
        // Value of 0 means the ship is not turning.
        return QString("Not turning");
    } else if (turn == 128) {
        // Value of 128 means the ROT is not defined.
        return QString("Not defined");
    } else if (turn == 127) {
        // Value of 127 means the ROT is more than 5degs/30s to right.
        return QString("More than 5") + QChar(0x00B0) + QString("/30s to right");
    } else if (turn == -127) {
        // Value of -127 means the ROT is more than 5degs/30s to left.
        return QString("More than 5") + QChar(0x00B0) + QString("/30s to left");
    }

    // Values -126 ... 126 mean something between 0 to 708 degrees per minute to right
    // (if ROT is positive) or to left (if ROT is negative).

    double degsPerMin = pow(turn / 4.733, 2.0);

    // E.g. "20.00 degs/min to right".
    return QString::number(degsPerMin, 'f', 2) +
            UnicodeDegreeSymbol + QString("/min") +
            QString(turn > 0 ? " to right" : " to left");
}

QString DataFormatter::speed(uint speed) {
    // Formats the speed over ground as a human-readable string.

    if (speed == 1023) {
        // Value of 1023 means the SOG is not defined.
        return QString("Not defined");
    }

    // Values 0 ... 1022 gives the speed over ground as knots with 0.1 precisions.
    // The SOG value therefore must be divided by 10 to get the actual value in knots.

    return QString::number(speed / 10.0, 'f', 1) + QString(" knots");
}

QString DataFormatter::course(uint course) {
    // Formats the course over ground as a human-readable string.

    if (course == 3600) {
        // Value of 3600 means the COG is not defined.
        return QString("Not defined");
    }

    // Values 0 ... 3599 givs the course over ground in degrees with 0.1 precision.
    // The COG value must therefore be divided by 10 to get the actual value in degs.

    return QString::number(course / 10.0, 'f', 1) + UnicodeDegreeSymbol;
}

QString DataFormatter::coord(const Coord &coord) {
    // Formats the coordinate as a pair of latitude & longitude pairs.

    if (coord.isUndefined()) return QString("Undefined");

    QString str;
    str += QChar(coord.lat() >= 0 ? '+' : '-');
    str += QString::number(coord.latDegrees(), 'f', 4);
    str += QChar(0x00B0);
    str += QChar(' ');
    str += QChar(coord.lon() >= 0 ? '+' : '-');
    str += QString::number(coord.lonDegrees(), 'f', 4);
    str += QChar(0x00B0);

    return str;
}

QString DataFormatter::distance(qreal distance) {
    // Formats the distance either as kilometers or meters depending on its magnitude.
    if (distance < 0.0) {
        return QString("Undefined");
    } else if (distance < 1.0) {
        return QString::number(distance * 1000.0, 'f', 2) + QString(" m");
    }
    return QString::number(distance, 'f', 2) + QString(" km");
}

QString DataFormatter::bearing(qreal bearing) {
    // Formats the bearing angle as is unless it's -1 (undefined).
    if (bearing < 0.0) {
        return QString("Undefined");
    }
    return QString::number(bearing, 'f', 2) + UnicodeDegreeSymbol;
}

QString DataFormatter::aspect(qreal aspect) {
    // Formats the aspect angle as is unless it's -1 (undefined).
    if (aspect < 0.0) {
        return QString("Undefined");
    }
    return QString::number(aspect, 'f', 1) + UnicodeDegreeSymbol;
}

QString DataFormatter::logMMSI(uint mmsi) {
    // Formats the MMSI as a string.
    return QString::number(mmsi);
}

QString DataFormatter::logCallSign(const QString &callSign) {
    // Returns the call sign as is.
    return callSign;
}

QString DataFormatter::logShipName(const QString &shipName) {
    // Returns the ship name as is.
    return shipName;
}

QString DataFormatter::logStatus(uint status) {
    // Formats the navigation status as a numeric code.
    return QString::number(status);
}

QString DataFormatter::logTurn(int turn) {
    // Formats the rate of turn as a decimal number except as '++'/'--' when
    // out of bounds.

    if (turn == 0) {
        return QString("0");
    } else if (turn == 128) {
        return QString();
    } else if (turn == 127) {
        return QString("++");
    } else if (turn == -127) {
        return QString("--");
    }
    double degsPerMin = pow(turn / 4.733, 2.0);
    return QString(turn > 0 ? "" : "-") + QString::number(degsPerMin, 'f', 5);
}

QString DataFormatter::logSpeed(uint speed) {
    // Formats the speed over ground as a decimal number unless it's undefined.
    if (speed == 1023) {
        return QString();
    }
    return QString::number(speed / 10.0, 'f', 1);
}

QString DataFormatter::logCourse(uint course) {
    // Formats the course over ground as a decimal number unless it's undefined.
    if (course == 3600) {
        return QString();
    }
    return QString::number(course / 10.0, 'f', 1);
}

QString DataFormatter::logLon(uint lon) {
    // Formats the longitude as a degrees.
    return QString::number(lon / 600000.0, 'f', 5);
}

QString DataFormatter::logLat(uint lat) {
    // Formats the latitude as degrees.
    return QString::number(lat / 600000.0, 'f', 5);
}

QString DataFormatter::logDistance(qreal distance) {
    // Formats the distance as a decimal number unless it's undefined.
    if (distance < 0.0) {
        return QString();
    }
    return QString::number(distance, 'f', 3);
}

QString DataFormatter::logBearing(qreal bearing) {
    // Formats the bearing angle as a decimal number unless it's undefined.
    if (bearing < 0.0) {
        return QString();
    }
    return QString::number(bearing, 'f', 3);
}

QString DataFormatter::logAspect(qreal aspect) {
    // Formats the aspect angle as a decimal number unless it's undefined.
    if (aspect < 0.0) {
        return QString();
    }
    return QString::number(aspect, 'f', 3);
}
```

File: dataformatter.h

```
/*
 * Pointing and Tracking Aid
 *
 * This software was written by Henri Häkkinen as part of his Bachelor's thesis for
 * the Danish Defence Acquistion and Logistics Organization during Spring 2011.
 *
 * dataformatter.h
 *
 * Class for formatting values received from the AIS into a more human-readable form.
 */

#ifndef DATAFORMATTER_H
#define DATAFORMATTER_H

#include <QString>

class Coord;

class DataFormatter {
public:
    // for the ship list
    static QString mmsi(uint mmsi);
    static QString callSign(const QString &callSign);
    static QString shipName(const QString &shipName);
    static QString status(uint status);
    static QString turn(int rot);
    static QString speed(uint speed);
    static QString course(uint course);
    static QString coord(const Coord &coord);
    static QString distance(qreal distance);
    static QString bearing(qreal bearing);
    static QString aspect(qreal aspect);

    // for the log file
    static QString logMMSI(uint mmsi);
    static QString logCallSign(const QString &callSign);
    static QString logShipName(const QString &shipName);
    static QString logStatus(uint status);
    static QString logTurn(int rot);
    static QString logSpeed(uint speed);
    static QString logCourse(uint course);
    static QString logLon(uint lon);
    static QString logLat(uint lat);
    static QString logDistance(qreal distance);
    static QString logBearing(qreal bearing);
    static QString logAspect(qreal aspect);
};

#endif // DATAFORMATTER_H
```

File: mailslot.cpp

```
/*
 * Pointing and Tracking Aid
 *
 * This software was written by Henri Häkkinen as part of his Bachelor's thesis for
 * the Danish Defence Acquistion and Logistics Organization during Spring 2011.
 *
 * mailslot.cpp
 *
 * Implementation of the Mailslot class.
 */

#include <QtConcurrentRun>
#include <QMessageBox>
#include "mailslot.h"
```

```
Mailslot::Mailslot(const QString &localName, QObject *parent) :
    QObject(parent),
    _localName(localName),
    _fullName("\\\\.\\mailslot\\" + localName)
{
    // Open a mailslot in the local computer and start running the poll-loop in a
    // background thread.
    _win32Handle = CreateMailslotA(_fullName.toLatin1().constData(), 0,
                                   MAILSLOT_WAIT_FOREVER, NULL);
    if (!_win32Handle) {
        QMessageBox::critical(0, "Error", "Could not create a mailslot\n" + _fullName);
    } else {
        QtConcurrent::run(this, &Mailslot::pollLoop);
    }
}

Mailslot::~Mailslot() {
    // Close the mailslot handle; will terminate the poll-loop.
    CloseHandle(_win32Handle);
}

void Mailslot::pollLoop() {
    // Poll messages from the mailslot in a loop. The loop runs until an error happens
    // while reading the mailslot. This function runs in a background thread.
    while (1) {
        if (!pollMessage()) break;
    }
}

bool Mailslot::pollMessage() {
    // Poll any awaiting messages from the mailslot. Return false if there were errors
    // while reading the mailslot in which case the poll-loop is terminated. This function
    // will wait and block until there is a message to read.
    //
    // Maxium length of a single message is 424 characters.

    char buffer[424];
    DWORD length = 0;

    if (!ReadFile(_win32Handle, buffer, 424, &length, 0)) {
//        qDebug() << "ReadFile failed.";
        return false;
    }

//    qDebug() << length << "bytes received from the mailslot";

    QByteArray message(buffer, length);
    emit messageReceived(QString(message));

    return true;
}
```

File: mailslot.h

```
/*
 * Pointing and Tracking Aid
 *
 * This software was written by Henri Häkkinen as part of his Bachelor's thesis for
 * the Danish Defence Acquistion and Logistics Organization during Spring 2011.
 *
 * mailslot.h
 *
 * Object-oriented Qt wrapper around the Windows Mailslot API.  Background thread is
 * started in the Mailslot constructor to listen for incoming messages.  Each time a
 * message is received, the dataReceived signal is emitted with the message passed in
 * as a function argument using a QByteArray object.
 *
 * A mailslot has a name, which is used by other processes to send messages into it.
 * The name has always the following format:
 *
 *   \\ComputerName\mailslot\MailslotName
 *
 * ComputerName is the name or the IP address of the host which opened the mailslot while
 * MailslotName is an arbitrary identifier of the mailslot.  To send a message to a local
 * mailslot, the process would use a dot instead of the host name.  To broadcast a message
 * to every mailslot in the Local Area Network, the sending process should use * as the
 * ComputerName.
 *
 * This class uses a Windows-specific API so therefore it is not portable outside the
 * Windows operating system.
 *
 * Windows Mailslot API reference:
 * - http://msdn.microsoft.com/en-us/library/aa365576(v=vs.85).aspx
 */

#ifndef MAILSLOT_H
#define MAILSLOT_H

#include <QObject>
#include <windows.h>

class Mailslot : public QObject {
    Q_OBJECT

public:
    Mailslot(const QString &localName, QObject *parent = 0);
    ~Mailslot();

    const QString &localName() const { return _localName; }
    const QString &fullName() const { return _fullName; }

signals:
    void messageReceived(const QString &data);

private:
    void pollLoop();
    bool pollMessage();

    QString _localName;
    QString _fullName;
    HANDLE _win32Handle;
};

#endif // MAILSLOT_H
```

File: main.cpp

```
/*
 * Pointing and Tracking Aid
 *
 * This software was written by Henri Häkkinen as part of his Bachelor's thesis for
 * the Danish Defence Acquistion and Logistics Organization during Spring 2011.
 *
 * main.cpp
 *
 * Application main entry-point.
 */

#include <QApplication>
#include <QDir>
#include "constants.h"
#include "mainwindow.h"

int main(int argc, char **argv) {
    // Initialize the application.
    QApplication a(argc, argv);
    QCoreApplication::setApplicationName(ApplicationName);
    QCoreApplication::setOrganizationName(OrganizationName);
    QCoreApplication::setOrganizationDomain(OrganizationDomain);

    // Make sure the log directory exists.
    QDir dir;
    dir.mkpath(LogDirectory);

    // Initialalize the main window.
    MainWindow w;
    w.show();

    // Run the application.
    return a.exec();
}
```

File: mainwindow.cpp

```
/*
 * Pointing and Tracking Aid
 *
 * This software was written by Henri Häkkinen as part of his Bachelor's thesis for
 * the Danish Defence Acquistion and Logistics Organization during Spring 2011.
 *
 * mainwindow.cpp
 *
 * Implementation of the MainWindow class.
 */

#include <QAction>
#include <QApplication>
#include <QCloseEvent>
#include <QMenu>
#include <QMenuBar>
#include <QMessageBox>
#include <QHeaderView>
#include <QSettings>
#include <QSplitter>
#include <QTableView>
#include <QTcpSocket>

#include "aisparser.h"
#include "constants.h"
#include "mailslot.h"
#include "mainwindow.h"
#include "mrsparser.h"
#include "radar.h"
#include "settings.h"
#include "shipmanager.h"

MainWindow::MainWindow(QWidget *parent) : QMainWindow(parent) {
    QSettings settings;
    // Create actions.
    _settingsAction = new QAction("&Settings...", this);
    _exitAction = new QAction("E&xit", this);
    _zoomInAction = new QAction("Zoom &in", this);
    _zoomOutAction = new QAction("Zoom &out", this);
    _aboutAppAction = new QAction("&About " + ApplicationName + "...", this);
    _aboutQtAction = new QAction("About Qt...", this);

    _zoomInAction->setShortcut(Qt::Key_Plus);
    _zoomOutAction->setShortcut(Qt::Key_Minus);

    // Create the menu.
    QMenu *fileMenu = menuBar()->addMenu("&File");
    QMenu *viewMenu = menuBar()->addMenu("&View");
    QMenu *helpMenu = menuBar()->addMenu("&Help");

    fileMenu->addAction(_settingsAction);
    fileMenu->addSeparator();
    fileMenu->addAction(_exitAction);
    viewMenu->addAction(_zoomInAction);
    viewMenu->addAction(_zoomOutAction);
    helpMenu->addAction(_aboutQtAction);
    helpMenu->addAction(_aboutAppAction);

    connect(_settingsAction, SIGNAL(triggered()), this, SLOT(settings()));
    connect(_aboutQtAction, SIGNAL(triggered()), qApp, SLOT(aboutQt()));
    connect(_aboutAppAction, SIGNAL(triggered()), this, SLOT(aboutApp()));
    connect(_exitAction, SIGNAL(triggered()), qApp, SLOT(quit()));

    // Create the main user interface.
    _splitter = new QSplitter(this);
    _shipList = new QTableView(this);
    _radar = new Radar(this);

    _splitter->setOrientation(Qt::Vertical);
    _splitter->addWidget(_radar);
    _splitter->addWidget(_shipList);

    _shipList->setShowGrid(false);
    _shipList->setAlternatingRowColors(true);
    _shipList->setSelectionMode(QAbstractItemView::SingleSelection);
    _shipList->setSelectionBehavior(QAbstractItemView::SelectRows);
    _shipList->setCornerButtonEnabled(false);

    setCentralWidget(_splitter);
    setWindowTitle(ApplicationName);
    setWindowIcon(QIcon(ApplicationIconName));
    readLayout();

    // Create the AIS parser.
    _tcpSocket = new QTcpSocket(this);
    _aisParser = new AISParser(_tcpSocket, this);

    // Create the MRS parser.
    _mailslot = new Mailslot(settings.value("MRS/mailslot", MRSMailslotName).toString(),
this);
    _mrsParser = new MRSParser(_mailslot, this);

    connect(_mrsParser, SIGNAL(beamAngleReceived(qreal)), _radar,
SLOT(setBeamAngle(qreal)));

    // Create the ship manager.
    _shipManager = new ShipManager(this);
    _shipList->setModel(_shipManager);
    _radar->setShipManager(_shipManager);

    connect(_shipManager, SIGNAL(shipSelectionChanged()), _radar, SLOT(repaint()));
    connect(_shipManager, SIGNAL(shipDatabaseChanged()), _radar, SLOT(repaint()));
    connect(_shipManager, SIGNAL(originChanged()), _radar, SLOT(repaint()));
    connect(_zoomInAction, SIGNAL(triggered()), _radar, SLOT(zoomIn()));
    connect(_zoomOutAction, SIGNAL(triggered()), _radar, SLOT(zoomOut()));
    connect(_shipList->selectionModel(),
SIGNAL(currentRowChanged(QModelIndex,QModelIndex)), _shipManager,
SLOT(selectShip(QModelIndex)));
    connect(_aisParser, SIGNAL(messageReceived(const AISMessage*)), _shipManager,
SLOT(handleMessage(const AISMessage*)));

    // Connect to the AIS server.
    _tcpSocket->connectToHost(settings.value("AIS/host", AISHostname).toString(),
                              settings.value("AIS/port", AISPort).toUInt(),
                              QIODevice::ReadOnly | QIODevice::Text);
}

void MainWindow::settings() {
    // Called in response of the "Settings..." menu item.  Displays a settings
    // dialog for the user.

    Settings settings;
    settings.exec();
}

void MainWindow::aboutApp() {
    // Called when the user selects "About Pointing and Tracking Aid" from the Help menu.
    // Displays a modal message box with a text explaining the application version, build
date
    // and the author.

    const QString text =
            "Version " + ApplicationVersion + "\n" +
            "Build Date " + __DATE__ + "\n\n" +
            "This software was written by Henri Häkkinen as part of his Bachelor's thesis
for\n" +
            "the Danish Defence Acquistion and Logistics Organization during Spring
2011.\n\n" +
            "Copyright (C) 2011 " + OrganizationName;

    QMessageBox::about(this, "About " + ApplicationName, text);
}

void MainWindow::closeEvent(QCloseEvent *event) {
    // Called by the Qt framework when the window is about to be closed; save
    // UI layout and quit.

    writeLayout();
    event->accept();
}

void MainWindow::writeLayout() {
    // Save the UI layout to the user preferences database.

    QSettings settings;
    settings.setValue("MainWindow/geometry", saveGeometry());
    settings.setValue("Splitter/state", _splitter->saveState());
    settings.setValue("ShipList/state", _shipList->horizontalHeader()->saveState());
}

void MainWindow::readLayout() {
    // Restore the UI layout from the user preferences database.
    QSettings settings;
    restoreGeometry(settings.value("MainWindow/geometry").toByteArray());
    _splitter->restoreState(settings.value("Splitter/state").toByteArray());
    _shipList->horizontalHeader()-
>restoreState(settings.value("ShipList/state").toByteArray());
}
```

File: mainwindow.h

```
/*
 * Pointing and Tracking Aid
 *
 * This software was written by Henri Häkkinen as part of his Bachelor's thesis for
 * the Danish Defence Acquistion and Logistics Organization during Spring 2011.
 *
 * mainwindow.h
 *
 * MainWindow class creates the user interface and acts as a central coordinator of
 * action.
 */

#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>

class QAction;
class QLineEdit;
class QSplitter;
class QTableView;
class QTcpSocket;
class QToolBar;

class Mailslot;
class Radar;
class ShipManager;
class AISParser;
class MRSParser;

class MainWindow : public QMainWindow {
    Q_OBJECT

public:
    MainWindow(QWidget *parent = 0);

private slots:
    void settings();
    void aboutApp();

private:
    void closeEvent(QCloseEvent *event);
    void writeLayout();
    void readLayout();

    QAction *_exitAction;
    QAction *_settingsAction;
    QAction *_zoomInAction;
    QAction *_zoomOutAction;
    QAction *_aboutAppAction;
    QAction *_aboutQtAction;

    QSplitter *_splitter;
    QTableView *_shipList;
    Radar *_radar;

    QTcpSocket *_tcpSocket;
    Mailslot *_mailslot;
    AISParser *_aisParser;
    MRSParser *_mrsParser;
    ShipManager *_shipManager;
};

#endif // MAINWINDOW_H
```

File: mrsparser.cpp

```
/*
 * Pointing and Tracking Aid
 *
 * This software was written by Henri Häkkinen as part of his Bachelor's thesis for
 * the Danish Defence Acquistion and Logistics Organization during Spring 2011.
 *
 * mrsparser.cpp
 *
 * Implementation of the MRSParser class.
 */

#include <QRegExp>
#include "mailslot.h"
#include "mrsparser.h"

MRSParser::MRSParser(Mailslot *mailslot, QObject *parent) :
    QObject(parent),
    _mailslot(mailslot)
{
    connect(_mailslot, SIGNAL(messageReceived(QString)), this,
SLOT(parseMessage(QString)));
}

void MRSParser::parseMessage(const QString &message) {
    // Called when a new message is received from the mailslot.

    static const QRegExp pattern("[0-9]{3}[.,][0-9]{2};");
    if (pattern.indexIn(message) != -1) {
        qreal angle = parseAngle(pattern.cap());
        emit beamAngleReceived(angle);
    }
}

qreal MRSParser::parseAngle(const QString &string) {
    // Called to parse an angle from the message.
    //
    // An angle is always formatted in the form "DDD.DD" where 'D' is a single
    // digit as the amount of degrees relatively to the reference angle.  For
    // example 90.15 degrees would be "090.15".

    if (string.length() < 6)
        return 0;

    int digit1 = string[0].toAscii() - '0';
    int digit2 = string[1].toAscii() - '0';
    int digit3 = string[2].toAscii() - '0';
    int digit4 = string[4].toAscii() - '0';
    int digit5 = string[5].toAscii() - '0';
    return digit1 * 100.0 + digit2 * 10.0 + digit3 * 1.0 + digit4 * 0.1 + digit5 * 0.01;
}
```

File: mrsparser.h

```
/*
 * Pointing and Tracking Aid
 *
 * This software was written by Henri Häkkinen as part of his Bachelor's thesis for
 * the Danish Defence Acquistion and Logistics Organization during Spring 2011.
 *
 * mrsparser.h
 *
 * Modular Radar System (MRS) message parser.  Parses message received from the MRS control
 * panel software using the Windows Mailslots IPC mechanism.
 */

#ifndef MRSPARSER_H
#define MRSPARSER_H

#include <QObject>

class Mailslot;

class MRSParser : public QObject {
    Q_OBJECT

public:
    MRSParser(Mailslot *mailslot, QObject *parent = 0);

signals:
    void beamAngleReceived(qreal angle);

private slots:
    void parseMessage(const QString &message);

private:
    qreal parseAngle(const QString &str);

    Mailslot *_mailslot;
};

#endif // MRSPARSER_H
```

File: payload.cpp

```
/*
 * Pointing and Tracking Aid
 *
 * This software was written by Henri Häkkinen as part of his Bachelor's thesis for
 * the Danish Defence Acquistion and Logistics Organization during Spring 2011.
 *
 * payload.cpp
 *
 * Implementation of Payload class.
 */

#include <QByteArray>
```

```
#include <QtDebug>
#include <cmath>
#include "payload.h"

static const QChar chartab[64] = {
    QChar('@'), QChar('A'), QChar('B'), QChar('C'), QChar('D'), QChar('E'), QChar('F'),
    QChar('G'), QChar('H'), QChar('I'), QChar('J'), QChar('K'), QChar('L'), QChar('M'),
    QChar('N'), QChar('O'), QChar('P'), QChar('Q'), QChar('R'), QChar('S'), QChar('T'),
    QChar('U'), QChar('V'), QChar('W'), QChar('X'), QChar('Y'), QChar('Z'), QChar('['),
    QChar('\\'), QChar(']'), QChar('^'), QChar('_'), QChar(' '), QChar('!'), QChar('"'),
    QChar('#'), QChar('$'), QChar('%'), QChar('&'), QChar('\''), QChar('('), QChar(')'),
    QChar('*'), QChar('+'), QChar(','), QChar('-'), QChar('.'), QChar('/'), QChar('0'),
    QChar('1'), QChar('2'), QChar('3'), QChar('4'), QChar('5'), QChar('6'), QChar('7'),
    QChar('8'), QChar('9'), QChar(':'), QChar(';'), QChar('<'), QChar('='), QChar('>'),
    QChar('?')
};

Payload::Payload(const QByteArray &data, int padBits) :
    _bits(data.length() * 6 - padBits)
{
    // Decode bits from the given series of bytes.

    for (int i = 0; i < data.count(); ++i) {
        char ch = data.at(i) - 48;
        if (ch >= 40) ch -= 8;
        for (int b = 0; b < 6; ++b) {
            if ((i * 6 + b) >= _bits.count())
                return;
            if (ch & (1 << (5 - b)))
                _bits.setBit(i * 6 + b);
        }
    }
}

bool Payload::extractBool(int index) const {
    // Returns the bit at the given index.

    if (index >= _bits.size()) {
        return false;
    }
    return _bits.testBit(index);
}

int Payload::extractInt(int start, int length) const {
    // Extracts a signed integer from the given range of payload bits.

    uint value = extractUInt(start, length);
    if (value & (1 << (length - 1))) {
        value = (uint) -(pow(2, length) - value);
    }
    return (int) value;
}

uint Payload::extractUInt(int start, int length) const {
    // Extracts an unsigned integer from the given range of payload bits.

    if ((start + length) >= _bits.size()) {
        return 0;
    }

    uint value = 0;
    for (int i = start; i < (start + length); ++i) {
        value <<= 1;
        if (_bits.testBit(i)) value |= 1;
    }
    //value &= ~(-1 << length);
    return value;
}

QString Payload::extractString(int start, int length) const {
    // Extracts a string from the given range of payload bits.

    if ((start + length) >= _bits.size()) {
        return QString();
    }

    QString value;
    for (int i = start; i < (start + length); i += 6) {
        uint ch = extractUInt(i, 6);
        if (ch == 0 || ch >= 64) break;
        value.append(chartab[ch]);
    }
    return value.trimmed();
}
```

File: payoad.h

```
/*
 * Pointing and Tracking Aid
 *
 * This software was written by Henri Häkkinen as part of his Bachelor's thesis for
 * the Danish Defence Acquistion and Logistics Organization during Spring 2011.
 *
 * payload.h
 *
 * Class handling the six-bit payload armoring.
 */

#ifndef PAYLOAD_H
#define PAYLOAD_H

#include <QBitArray>

class QByteArray;

class Payload {
public:
    Payload(const QByteArray &data, int padBits = 0);

    bool extractBool(int index) const;
    int extractInt(int start, int length) const;
    uint extractUInt(int start, int length) const;
    QString extractString(int start, int length) const;

private:
    QBitArray _bits;
};

#endif // PAYLOAD_H
```

File: radar.cpp

```
/*
 * Pointing and Tracking Aid
 *
 * This software was written by Henri Häkkinen as part of his Bachelor's thesis for
 * the Danish Defence Acquistion and Logistics Organization during Spring 2011.
 *
 * radar.cpp
 *
 * Implementation of the Radar class.
 */

#include <QFont>
#include <QLinearGradient>
#include <QPaintEvent>
#include <QPainter>
#include <QPainterPath>
#include <cmath>

#include "constants.h"
#include "radar.h"
#include "ship.h"
#include "shipmanager.h"

Radar::Radar(QWidget *parent) :
    QWidget(parent),
    _beamAngle(0.0),
    _zoom(RadarDefaultZoom),
    _shipManager(0)
{ /* nothing to do here */ }

void Radar::paintEvent(QPaintEvent */*event*/) {
    // Called by the framework when the radar map needs repainting.

    int size = qMin(width(), height());
    if (size > 0) {
        QPainter painter(this);
        painter.translate(width() / 2.0, height() / 2.0);
        painter.scale(size / 200.0, size / 200.0);
        painter.setRenderHint(QPainter::Antialiasing, true);
        paintRadar(painter);
        paintBeam(painter);
        paintShips(painter);
    }
}

void Radar::paintRadar(QPainter &painter) {
    // Paints the radar map.

    QLinearGradient gradient(-100, 100, 100, -100);
    gradient.setColorAt(0, QColor(0, 10, 0));
    gradient.setColorAt(1, QColor(0, 120, 0));

    // background color
    QPainterPath path;
    path.addEllipse(-100, -100, 200, 200);
    painter.fillPath(path, QBrush(gradient));

    // coordinate axes
    painter.setPen(QPen(QColor(0, 200, 0)));
    painter.drawLine(0, -100, 0, 100);
    painter.drawLine(-100, 0, 100, 0);
    painter.drawEllipse(-100, -100, 200, 200);

    // coordinate system origin
    painter.setFont(QFont("Courier New", 4));
    QString lat = QString::number(_shipManager->origin().latDegrees(), 'f', 4);
    QString lon = QString::number(_shipManager->origin().lonDegrees(), 'f', 4);
    painter.drawText(0, 0, QString("(%1, %2)").arg(lat).arg(lon));
}

void Radar::paintBeam(QPainter &painter) {
    // Paints the radar beam.

    qreal radians = (_beamAngle - 90.0) * M_PI/180.0;
    QPainterPath beamPath;
    beamPath.moveTo(0, 0);
    beamPath.lineTo(100.0*cos(radians - RadarBeamAngle), 100.0*sin(radians -
RadarBeamAngle));
    beamPath.lineTo(100.0*cos(radians + RadarBeamAngle), 100.0*sin(radians +
RadarBeamAngle));
    beamPath.closeSubpath();
    painter.fillPath(beamPath, QColor(0, 255, 0, 70));
}

void Radar::paintShips(QPainter &painter) {
    // Paints ships on the radar map.

    painter.scale(1.0, -1.0);
    for (int t = 0; t < _shipManager->shipCount(); ++t) {
        Ship *ship = _shipManager->shipAt(t);
        qreal x = (ship->coord().x() - _shipManager->origin().x()) * (100.0 / _zoom);
        qreal y = (ship->coord().y() - _shipManager->origin().y()) * (100.0 / _zoom);
        if ((x*x + y*y) <= 10000) {
            QColor color;
            color = ((ship != _shipManager->selectedShip()) ? ShipColorRegular :
ShipColorSelected);
            paintShip(painter, x, y, ship->course()/10.0, color);
        }
    }
}

void Radar::paintShip(QPainter &painter, qreal dx, qreal dy, qreal course, const QColor
&color) {
    // Paints a ship on the radar map.

    static const QPointF shape[] = {
        QPointF(ShipSize*cos( 0.436332313), ShipSize*sin( 0.436332313)),
        QPointF(ShipSize*cos( 3.141592650), ShipSize*sin( 3.141592650)),
        QPointF(ShipSize*cos(-0.436332313), ShipSize*sin(-0.436332313))
    };

    QTransform transform = painter.worldTransform();
    painter.setBrush(color);
    painter.setPen(color);
    painter.translate(dx, dy);
    painter.rotate(-90.0 - course);
    painter.drawPolygon(shape, 3);
    painter.setWorldTransform(transform);
}
```

File: radar.h

```
/*
 * Pointing and Tracking Aid
 *
 * This software was written by Henri Häkkinen as part of his Bachelor's thesis for
 * the Danish Defence Acquistion and Logistics Organization during Spring 2011.
 *
 * radar.h
 *
 * Radar user-interface element.
 */

#ifndef RADAR_H
#define RADAR_H

#include <QWidget>
#include "coord.h"

class ShipManager;

class Radar : public QWidget {
    Q_OBJECT

public:
    Radar(QWidget *parent = 0);

    QSize sizeHint() const { return QSize(400, 400); }
    qreal beamAngle() const { return _beamAngle; }
    qreal zoom() const { return _zoom; }

    ShipManager *shipManager() const { return _shipManager; }
    void setShipManager(ShipManager *sm) { _shipManager = sm; }

public slots:
    void setBeamAngle(qreal angle) { _beamAngle = angle; repaint(); }
    void zoomIn() { _zoom -= 0.001; repaint(); }
    void zoomOut() { _zoom += 0.001; repaint(); }

private:
    void paintEvent(QPaintEvent *event);
    void paintRadar(QPainter &painter);
    void paintBeam(QPainter &painter);
    void paintShips(QPainter &painter);
    void paintShip(QPainter &painter, qreal dx, qreal dy, qreal course, const QColor
&color);

    qreal _beamAngle;
    qreal _zoom;
    ShipManager *_shipManager;
};

#endif // RADAR_H
```

File: settings.cpp

```
/*
 * Pointing and Tracking Aid
 *
 * This software was written by Henri Häkkinen as part of his Bachelor's thesis for
 * the Danish Defence Acquistion and Logistics Organization during Spring 2011.
 *
 * settings.cpp
 *
 * Implementation of Settings class.
 */

#include <QFormLayout>
#include <QHBoxLayout>
#include <QLineEdit>
#include <QPushButton>
#include <QSettings>
```

```
#include <QRegExp>

#include <QRegExpValidator>
#include <QDoubleValidator>
#include <QIntValidator>

#include "constants.h"
#include "settings.h"

Settings::Settings(QWidget *parent) :
    QDialog(parent)
{
    QSettings settings;

    QFormLayout *formLayout = new QFormLayout(this);
    QHBoxLayout *hboxLayout = new QHBoxLayout(this);

    _mailslotName = new QLineEdit(settings.value("MRS/mailslot",
QVariant(MRSMailslotName)).toString(), this);
    _aisHost = new QLineEdit(settings.value("AIS/host", QVariant(AISHostname)).toString(),
this);
    _aisPort = new QLineEdit(settings.value("AIS/port", QVariant(AISPort)).toString(),
this);
    _latitude = new QLineEdit(settings.value("Radar/latitude",
QVariant(RadarDefaultLatitude)).toString(), this);
    _longitude = new QLineEdit(settings.value("Radar/longitude",
QVariant(RadarDefaultLongitude)).toString(), this);

    _mailslotName->setValidator(new QRegExpValidator(QRegExp("[a-zA-Z0-9_-]+"), this));
    _aisHost->setValidator(new QRegExpValidator(QRegExp("[a-zA-Z0-9_-.]+"), this));
    _aisPort->setValidator(new QIntValidator(1, 65535, this));
    _latitude->setValidator(new QDoubleValidator(-90.0, 90.0, 4, this));
    _longitude->setValidator(new QDoubleValidator(-180.0, 180.0, 4, this));

    QPushButton *okButton = new QPushButton("Ok", this);
    QPushButton *cancelButton = new QPushButton("Cancel", this);

    hboxLayout->addWidget(okButton);
    hboxLayout->addWidget(cancelButton);

    formLayout->addRow("MRS mailslot name", _mailslotName);
    formLayout->addRow("AIS host/ip address", _aisHost);
    formLayout->addRow("AIS port number", _aisPort);
    formLayout->addRow("Default latitude", _latitude);
    formLayout->addRow("Default longitude", _longitude);
    formLayout->addRow(hboxLayout);

    setLayout(formLayout);
    setWindowTitle("Settings");

    connect(okButton, SIGNAL(clicked()), this, SLOT(accept()));
    connect(cancelButton, SIGNAL(clicked()), this, SLOT(reject()));
}

void Settings::accept() {
    // Called when the Ok button is pressed.  Writes the settings to the user
    // preferences database.

    QSettings settings;
    settings.setValue("MRS/mailslot", _mailslotName->text());
    settings.setValue("AIS/host", _aisHost->text());
    settings.setValue("AIS/port", _aisPort->text());
    settings.setValue("Radar/latitude", _latitude->text());
    settings.setValue("Radar/longitude", _longitude->text());
    QDialog::accept();
}
```

File: settings.h

```
/*
 * Pointing and Tracking Aid
 *
 * This software was written by Henri Häkkinen as part of his Bachelor's thesis for
 * the Danish Defence Acquistion and Logistics Organization during Spring 2011.
 *
 * settings.h
 *
 * Settings dialog window.
 */

#ifndef SETTINGS_H
#define SETTINGS_H

#include <QDialog>

class QLineEdit;

class Settings : public QDialog {
    Q_OBJECT

public:
    Settings(QWidget *parent = 0);

public slots:
    void accept();

private:
    QLineEdit *_mailslotName;
    QLineEdit *_aisHost;
    QLineEdit *_aisPort;
    QLineEdit *_latitude;
    QLineEdit *_longitude;
};

#endif // SETTINGS_H
```

File: ship.cpp

```
/*
 * Pointing and Tracking Aid
 *
 * This software was written by Henri Häkkinen as part of his Bachelor's thesis for
 * the Danish Defence Acquistion and Logistics Organization during Spring 2011.
 *
 * ship.cpp
 *
 * Implementation of Ship class.
 */

#include "aismessages.h"
#include "ship.h"
#include "shipmanager.h"
```

```
Ship::Ship(ShipManager *parent) :
    QObject(parent),
    _logged(false)
{
    connect(this, SIGNAL(dataChanged(Ship*)), parent, SLOT(logShip(Ship*)));
}

Ship::Ship(const PositionReport *message, ShipManager *parent) :
    QObject(parent),
    _logged(false),
    _mmsi(message->mmsi()),
    _status(message->status()),
    _turn(message->turn()),
    _speed(message->speed()),
    _course(message->course()),
    _coord(message->coord())
{
    connect(this, SIGNAL(dataChanged(Ship*)), parent, SLOT(logShip(Ship*)));
}

Ship::Ship(const ShipVoyageData *message, ShipManager *parent) :
    QObject(parent),
    _logged(false),
    _callSign(message->callSign()),
    _shipName(message->shipName())
{
    connect(this, SIGNAL(dataChanged(Ship*)), parent, SLOT(logShip(Ship*)));
}

void Ship::update(const PositionReport *message) {
    // Updates state from the given AIS message.  If logging is enabled for this ship
    // emits dataChanged signal which will be processed by ShipManager's logShip slot.

    _mmsi = message->mmsi();
    _status = message->status();
    _turn = message->turn();
    _speed = message->speed();
    _course = message->course();
    _coord = message->coord();

    if (_logged) {
        emit dataChanged(this);
    }
}

void Ship::update(const ShipVoyageData *message) {
    // Updates state from the given AIS message.  If logging is enabled for this ship
    // emits dataChanged signal which will be processed by ShipManager's logShip slot.

    _callSign = message->callSign();
    _shipName = message->shipName();

    if (_logged) {
        emit dataChanged(this);
    }
}
```

File: ship.h

```
/*
 * Pointing and Tracking Aid
 *
 * This software was written by Henri Häkkinen as part of his Bachelor's thesis for
 * the Danish Defence Acquistion and Logistics Organization during Spring 2011.
 *
 * ship.h
 *
 * Encapsulates data for a single ship.
 */

#ifndef SHIP_H
#define SHIP_H

#include <QObject>
#include "coord.h"

class ShipManager;
class PositionReport;
class ShipVoyageData;

class Ship : public QObject {
    Q_OBJECT

public:
    Ship(ShipManager *parent);
    Ship(const PositionReport *message, ShipManager *parent);
    Ship(const ShipVoyageData *message, ShipManager *parent);

    bool    isLogged() const         { return _logged; }
    void    setLogged(bool logged)   { _logged = logged; }
    uint    mmsi() const             { return _mmsi; }
    QString callSign() const         { return _callSign; }
    QString shipName() const         { return _shipName; }
    uint    status() const           { return _status; }
    int     turn() const             { return _turn; }
    uint    speed() const            { return _speed; }
    uint    course() const           { return _course; }
    int     lon() const              { return _coord.lon(); }
    int     lat() const              { return _coord.lat(); }
    Coord   coord() const            { return _coord; }

    void update(const PositionReport *message);
    void update(const ShipVoyageData *message);

signals:
    void dataChanged(Ship *ship);

private:
    bool _logged;                 // Ship is being logged?
    uint _mmsi;                   // Mobile Marine Service Identifier
    QString _callSign;            // Call Sign
    QString _shipName;            // Ship Name
    uint _status;                 // Navigation Status
    int _turn;                    // Rate of Turn
    uint _speed;                  // Speed Over Ground
    uint _course;                 // Course Over Ground
    Coord _coord;                 // Latitude & Longitude
};

#endif // SHIP_H
```

File: shipmanager.cpp

```
/*
 * Pointing and Tracking Aid
 *
 * This software was written by Henri Häkkinen as part of his Bachelor's thesis for
 * the Danish Defence Acquistion and Logistics Organization during Spring 2011.
 *
 * shipmanager.cpp
 *
 * Implementation of the ShipManager class.
 */

#include <QtDebug>
#include <QDate>
#include <QDateTime>
#include <QDir>
#include <QFile>
#include <QSettings>
#include <QTextStream>

#include "aismessages.h"
#include "constants.h"
#include "dataformatter.h"
#include "ship.h"
#include "shipmanager.h"

enum {
    // Hardcoded column indices:

    COLUMN_LOG,         // Log checkbox
    COLUMN_MMSI,        // MMSI
    COLUMN_CALLSIGN,    // Call sign
    COLUMN_SHIPNAME,    // Ship name
    COLUMN_STATUS,      // Navigatin Status
    COLUMN_COORDS,      // Coordinates
    COLUMN_ROT,         // Rate of Turn
    COLUMN_SOG,         // Speed Over Ground
    COLUMN_COG,         // Course Over Ground
    COLUMN_DISTANCE,    // Distance
    COLUMN_BEARING,     // Bearing
    COLUMN_ASPECT,      // Aspect

    COLUMNCOUNT         // Total number of columns
};

ShipManager::ShipManager(QObject *parent) :
    QAbstractTableModel(parent),
    _selectedShip(0)
{
    QSettings settings;

    // Read the default origin from the user preferences database.
    _origin = Coord(settings.value("Radar/latitude",
RadarDefaultLatitude).toDouble()*600000,
                    settings.value("Radar/longitude",
RadarDefaultLongitude).toDouble()*600000);
}

int ShipManager::rowCount(const QModelIndex &/*parent*/) const {
    // Returns the number of rows in this item model; that is, the number of ships received
    // so far.

    return _ships.count();
}

int ShipManager::columnCount(const QModelIndex &/*parent*/) const {
    // Returns the number of columns in this item model; that is, the number of data
    // elements per ship.
    return COLUMNCOUNT;
}

Qt::ItemFlags ShipManager::flags(const QModelIndex &index) const {
    // Returns the set of flags for the given model index.  This will make the first column
    // (log) to be displayed as a checkbox.

    if (index.column() == COLUMN_LOG) {
        return Qt::ItemIsUserCheckable | Qt::ItemIsEnabled | Qt::ItemIsSelectable;
    }
    return Qt::ItemIsEnabled | Qt::ItemIsSelectable;
}

QVariant ShipManager::headerData(int section, Qt::Orientation orientation, int role) const
{
    // Called to get the labels etc. in the horizontal and the vertical header views.
    //
    // We are only interested in the top horizontal header view without any tooltips
    // or stuff like that.

    if ((orientation == Qt::Horizontal) && (role == Qt::DisplayRole)) {
        switch (section) {
        case COLUMN_LOG:        return QVariant("Log");
        case COLUMN_MMSI:       return QVariant("MMSI");
        case COLUMN_CALLSIGN:   return QVariant("Call sign");
        case COLUMN_SHIPNAME:   return QVariant("Ship name");
        case COLUMN_COORDS:     return QVariant("Coordinates");
        case COLUMN_STATUS:     return QVariant("Navigation Status");
        case COLUMN_ROT:        return QVariant("Rate of Turn");
        case COLUMN_SOG:        return QVariant("Speed Over Ground");
        case COLUMN_COG:        return QVariant("Course Over Ground");
        case COLUMN_DISTANCE:   return QVariant("Distance");
        case COLUMN_BEARING:    return QVariant("Bearing");
        case COLUMN_ASPECT:     return QVariant("Aspect");
        }
    }

    return QVariant();
}

QVariant ShipManager::data(const QModelIndex &index, int role) const {
    // Called to get the data element for each cell.

    if (!index.isValid())
        return QVariant();

    if (index.row() >= _ships.count())
        return QVariant();

    if (index.column() > COLUMNCOUNT)
        return QVariant();

    if (role == Qt::DisplayRole) {
        Ship *ship = _ships[index.row()];
        switch (index.column()) {
        case COLUMN_MMSI:       return QVariant(DataFormatter::mmsi(ship->mmsi()));
        case COLUMN_CALLSIGN:   return QVariant(DataFormatter::callSign(ship->callSign()));
        case COLUMN_SHIPNAME:   return QVariant(DataFormatter::shipName(ship->shipName()));
        case COLUMN_COORDS:     return QVariant(DataFormatter::coord(ship->coord()));
        case COLUMN_STATUS:     return QVariant(DataFormatter::status(ship->status()));
        case COLUMN_ROT:        return QVariant(DataFormatter::turn(ship->turn()));
        case COLUMN_SOG:        return QVariant(DataFormatter::speed(ship->speed()));
        case COLUMN_COG:        return QVariant(DataFormatter::course(ship->course()));
        case COLUMN_DISTANCE:   return
QVariant(DataFormatter::distance(Coord::distance(_origin, ship->coord())));
```

```
        case COLUMN_BEARING:    return
QVariant(DataFormatter::bearing(Coord::bearing(_origin, ship->coord())));
        case COLUMN_ASPECT:     return
QVariant(DataFormatter::aspect(Coord::aspect(_origin, ship->coord(), ship->course())));
        }
    } else if (role == Qt::CheckStateRole) {
        if (index.column() == COLUMN_LOG) {
            return QVariant(_ships[index.row()]->isLogged() ? Qt::Checked : Qt::Unchecked);
        }
    } else if (role == Qt::TextAlignmentRole) {
        return QVariant(Qt::AlignHCenter);
    }

    return QVariant();
}

bool ShipManager::setData(const QModelIndex &index, const QVariant &value, int role) {
    // Called to alter the data in a cell.  Only the "log" column is allowed to be edited.

    if (!index.isValid())
        return false;

    if (index.row() >= _ships.count())
        return false;

    if (index.column() != COLUMN_LOG)
        return false;

    if (role == Qt::CheckStateRole) {
        _ships[index.row()]->setLogged(value.toBool());
        return true;
    }

    return false;
}

void ShipManager::handleMessage(const AISMessage *message) {
    // Handle a message received from the AIS.

    if (!message->isAIVDM()) {
        // AIVDO message.
        if (message->type() >= 1 && message->type() <= 3) {
            updateLocal(qobject_cast<const PositionReport *>(message));
        }
    } else {
        // AIVDM message.
        if (message->type() >= 1 && message->type() <= 3) {
            updateRemote(qobject_cast<const PositionReport *>(message));
        } else if (message->type() == 5) {
            updateRemote(qobject_cast<const ShipVoyageData *>(message));
        }
    }
}

void ShipManager::selectShip(const QModelIndex &index) {
    // Called when the user selects a row in the ship list.

    if (!index.isValid())
        return;

    if (index.row() >= _ships.count())
        return;

    Ship *ship = _ships.at(index.row());
    if (ship != _selectedShip) {
        _selectedShip = ship;
        emit shipSelectionChanged();
    }
}

void ShipManager::logShip(Ship *ship) {
    // Called to log a ship into a file.
    //
    // The log file is a line-based plain text file in which each log entry is appened to
    // end.  Log files are stored in the log directory (the LogDirectory compile-time
    // constant) and are named by the current date.
    //
    // The log file format is intended to be machine readable. A single line in the log
    // file looks like the following:
    //   [DATE]MMSI;CALLSIGN;SHIPNAME;NAVSTATUS;LAT;LON;ROT;SOG;COG;DISTANCE;BEARING;ASPECT
    //
    // The [DATE] is an ISO 8601 standard conforming timestamp.

    QString filename = LogDirectory + QDir::separator() +
QDate::currentDate().toString(Qt::ISODate)
                       + LogFileExtension;

    QFile file(filename);
    if (!file.open(QIODevice::WriteOnly | QIODevice::Append | QIODevice::Text))
        return;

    QTextStream output(&file);
    output << "[" << qPrintable(QDateTime::currentDateTime().toString(Qt::ISODate)) << "]"
           << qPrintable(DataFormatter::logMMSI(ship->mmsi()) << ";"
           << qPrintable(DataFormatter::logCallSign(ship->callSign())) << ";"
           << qPrintable(DataFormatter::logShipName(ship->shipName())) << ";"
           << qPrintable(DataFormatter::logStatus(ship->status())) << ";"
           << qPrintable(DataFormatter::logLat(ship->lat())) << ";"
           << qPrintable(DataFormatter::logLon(ship->lon())) << ";"
           << qPrintable(DataFormatter::logTurn(ship->turn())) << ";"
           << qPrintable(DataFormatter::logSpeed(ship->speed())) << ";"
           << qPrintable(DataFormatter::logCourse(ship->course())) << ";"
           << qPrintable(DataFormatter::logDistance(Coord::distance(_origin, ship-
>coord()))) << ";"
           << qPrintable(DataFormatter::logBearing(Coord::bearing(_origin, ship->coord())))
<< ";"
           << qPrintable(DataFormatter::logAspect(Coord::aspect(_origin, ship->coord(),
ship->course()))) << endl;
}

void ShipManager::updateLocal(const PositionReport *message) {
    // Called to handle a Position Report for the local AIS station.

    if (!message->coord().isUndefined()) {
        _origin = message->coord();
        emit originChanged();
        emit dataChanged(index(0, COLUMN_DISTANCE), index(_ships.count(), COLUMN_ASPECT));
    }
}

void ShipManager::updateRemote(const PositionReport *message) {
    // Called to handle a Position Report for a remote vessel.

    int idx;
    Ship *ship = findShip(message->mmsi(), &idx);
    if (ship) {
        ship->update(message);
        emit dataChanged(index(idx, 0), index(idx, COLUMNCOUNT));
        emit shipDatabaseChanged();
    } else {
        beginInsertRows(QModelIndex(), _ships.count(), _ships.count() + 1);
        _ships.append(new Ship(message, this));
        endInsertRows();
        emit shipDatabaseChanged();
    }
//    qDebug() << "Position Report with MMSI" << message->mmsi();
}

void ShipManager::updateRemote(const ShipVoyageData *message) {
    // Called to handle a Ship And Voyage Data for a remote vessel.

    int idx;
    Ship *ship = findShip(message->mmsi(), &idx);
    if (ship) {
        ship->update(message);
        emit dataChanged(index(idx, 0), index(idx, COLUMNCOUNT));
        emit shipDatabaseChanged();
    } else {
        beginInsertRows(QModelIndex(), _ships.count(), _ships.count() + 1);
        _ships.append(new Ship(message, this));
        endInsertRows();
        emit shipDatabaseChanged();
    }
//    qDebug() << "Ship and Voyage Data with MMSI" << message->mmsi();
}

Ship *ShipManager::findShip(const uint &mmsi, int *shipIndex) const {
    // Find a ship for the specified MMSI from the ship list database.  If 'shipIndex'
    // is not 0, it will contain the ship's index in the database.

    for (int i = 0; i < _ships.count(); ++i) {
        Ship *ship = _ships[i];
        if (ship->mmsi() == mmsi) {
            if (shipIndex) *shipIndex = i;
            return ship;
        }
    }
    return 0;
}
```

File: shipmanager.h

```
/*
 * Pointing and Tracking Aid
 *
 * This software was written by Henri Häkkinen as part of his Bachelor's thesis for
 * the Danish Defence Acquistion and Logistics Organization during Spring 2011.
 *
 * shipmanager.h
 *
 * ShipManager manages a list of stations (eg. ships and other recognized identities)
 * within the AIS area.  The class implements QAbstractTableModel which allows it to be
 * used as an item model for a QTableView.
 */

#ifndef SHIPMANAGER_H
#define SHIPMANAGER_H

#include <QAbstractTableModel>
#include <QList>
#include "coord.h"

class AISMessage;
class PositionReport;
class ShipVoyageData;
class Ship;

class ShipManager : public QAbstractTableModel {
    Q_OBJECT

public:
    ShipManager(QObject *parent = 0);

    Coord origin() const { return _origin; }
    Ship *selectedShip() const { return _selectedShip; }
    Ship *shipAt(int i) const { return _ships.at(i); }
    int shipCount() const { return _ships.count(); }

    // QAbstractTableModel overrides.
    int rowCount(const QModelIndex &parent = QModelIndex()) const;
    int columnCount(const QModelIndex &parent = QModelIndex()) const;
    Qt::ItemFlags flags(const QModelIndex &index) const;
    QVariant headerData(int section, Qt::Orientation orientation, int role =
Qt::DisplayRole) const;
    QVariant data(const QModelIndex &index, int role = Qt::DisplayRole) const;
    bool setData(const QModelIndex &index, const QVariant &value, int role);

signals:
    void shipSelectionChanged();
    void shipDatabaseChanged();
    void originChanged();

public slots:
    void handleMessage(const AISMessage *message);
    void selectShip(const QModelIndex &index);
    void logShip(Ship *ship);

private:
    void updateLocal(const PositionReport *message);
    void updateRemote(const PositionReport *message);
    void updateRemote(const ShipVoyageData *message);
    Ship *findShip(const uint &mmsi, int *shipIndex) const;

    Coord _origin;
    QList<Ship *> _ships;
    Ship *_selectedShip;
};

#endif // SHIPMANAGER_H
```