31/5-2002

Java-framework for GPS

ITU F2002

Made by

Henrik Aasted Sørensen, haas@itu.dk

Supervised by

Thomas Hildebrandt, hilde@itu.dk

1. Introduction 4
2. Aim and background 5
2.1Aim 5
2.2TheGPS -system 5
2.2.1HowGPS -unitsgettheirposition 6
2.3TheGarminProtocol 6
2.3.1Physical 7
2.3.2Link 7
2.3.3Application 8
3. Problem analysis and design considerations 9
3.1GeographicUnitsforgeneralGPS -classes 9
3.2GPS -informationpropagation 9
3.3GPS -connection 10
4. User's Guide 10
4.1Prerequisites .... .10
4.2Packagestructure 11
4.3UsingtheGPS -librarywithaGarmin -GPS 11
4.4UsingtheAreaAlarm -service 13
5. Developer's Guide 14
5.1Extendingdk.itu.haas.GPS.GPS 14
6. Implementation details 18
6.1GeneralGPS -classes 18
6.2Garmin -implementation 18
6.2.1Basiccommunication 18
6.2.2TheGarminPacket -class 19
6.2.3Thedispatchingthread 20
6.2.4TheGarminListen er-interface 20
7. Testing 21
8. Conclusion 21
9. References 22
10. Appendix 23
10.1dk.itu.haas.GPS –package 23
10.1.1GPS 23
10.1.2IGPSlistener 25
10.1.3IwaypointListener 26
10.1.4ItransferListener 26
10.1.5IPosition 26
10.1.6ITime 26

10.1.7IDate 27
10.1.8IWaypoint 27
10.1.9Position 27
10.1.10PositionDegrees 28
10.1.11PositionRadians 28
10.1.12FeatureNotSupportedException 29

10.2dk.itu.haas.GPS.Garmin –package 30
10.2.1GarminGPS 30
10.2.2GarminPacket 33
10.2.3Positio nDataPacket 37
10.2.4PVTDataPacket 38
10.2.5TimeDataPacket 40
10.2.6WaypointDataPacket 41
10.2.7ProductDataPacket 43
10.2.8ProtocolDataPacket 44
10.2.9RecordsPacket 44
10.2.10GarminInputStream 45
10.2.11GarminOutputStream 46
10.2.12GarminListener 46
10.2.13InvalidPacketException 47
10.2.14PacketNotRecognizedException 47
10.2.15UnknownPacketException 48

10.3dk.itu.haas.GPS.services –package 48
10.3.1AreaAlarm 48
10.3.2IAlarmListener 50

10.4dk.itu.haas.GPS.examples –package 50
10.4.1ConnectionTest 50
10.4.2AreaAlarmDemo 52

1.Introduction

This report was written in May 2002 in a 4-week-project at the IT University of Copenhagen.

The aim of the project was to create a Java-framework that would allow people to use a GPS-unit in their programs. The framework was developed using a Garmin Etrex-GPS, although the design of the framework was made with extension to other GPS-types in mind.

I would like to thank Michael Frederiksen for supplying me with a connector cable for the GPS, and thus allowing me to start the project on time.

The framework and its documentation can be found at the project's homepage[1].

Henrik Aasted Sørensen

[1] http://www.it-c.dk/~haas/gps

2.Aimandbackground

2.1Aim

As electronic applications become more and more mobile, the desire to be able to tell exactly where a user is located grows. This will open up to what is called "location based services", which is expected to be the Next Big Thing ™. It seems likely that GPS is the technology that will make these technologies a reality. As GPS-chips are getting cheaper and cheaper, they are put into more and more devices. Mobile phones will probably be the next consumer-device to become GPS-enabled.
At the moment, though, most GPS-devices are stand-alone entities. Usually they contain some interface that allows them to be used from other electronic devices.

The aim of this project is to create a Java-framework that enables programmers to easily interface with these GPS-units and add GPS-capabilities to their programs. The framework should make a clear boundary between the communication-protocol of the GPS-unit and the actual GPS-functionality, ie. hide the protocol-details from the user of the framework. The framework should make it possible to change from one type of GPS to another with a minimum of code-change.

The framework should not be a framework for making an interface-program for the GPS, ie. a program that can transfer waypoints, routes and maps back and forth between the GPS and the host. It should rather be a framework that allows the essential GPS-information to be transferred from the GPS to the host, thus making it possible to experiment with "location based services" without having to invest in the newest mobile phone.

In this report, the word "host" will, in accordance with the Garmin-protocol specification, refer to the device communicating with the GPS-unit. The terms "the Java-library" or "the framework" refers to the collection of classes in the package `dk.itu.haas.GPS`.

2.2TheGPS -system

The GPS (Global Positioning System) was created by the US Department of Defense for military applications. It consists of 24 satellites in 12-hours orbits around the earth. The last of the satellites was launched in March 1994. The satellites' orbits are calculated so that at least 5 satellites can always be seen from any point on the Earth.

The system was initially made for military purposes, but has since found it's way into many civilian applications. Today GPS-units can be found in cars, boats, planes and many other systems. GPS-receivers can be found in one-chip-systems, which are

rapidly falling in price. It's therefore expected that GPS will be an integral part of many electronic applications in the future. "Location-based services" has rapidly become a buzz-word in the mobile industry.

2.2.1HowGPS -unitsgettheirposition

5 ground stations constantly measure and supervise the satellites. When a ground station discovers an error between the satellites calculated position, which all GPS-units contain in an almanac, and it's real position, the correction is transmitted to the satellite, which in turn transmits it to the GPS-units on the ground.

4 satellites are used to find the position of a GPS-unit. The GPS calculates its distance from 3 of the satellites. This results in 3 spheres upon which the GPS-unit can be located. When all 3 spheres are taken into account, the unit will end up with 2 potential positions. Usually one of them is either several hundreds of miles above the Earth's surface or has a very unrealistic velocity, and can thus easily be ruled out. The trick to measuring the distance from the satellite is to calculate the delay in the satellite's signal very accurately. This is what the 4th satellite is for. All the GPS-satellites contain an atomic clock. This allows anyone who knows the signal transmitted, and has the exact same timing as the satellite, to calculate the distance to the satellite. The problem is that atomic clocks in GPS-units is a requirement that would render the technology expensive and not as widely deployed as it is today. The 4th satellite is used to be able to make an assesement of how wrong the GPS-unit's clock is compared to the satellite's. This is done by measuring the distance to the 4th satellite and adjusting the result until a point is found where all 4 satelites' distances meet. The difference calculated is the difference between the unit's and the satelite's clock. As a result of this, a GPS-unit actually contains a very good approximation to an atomic clock.

2.3TheGarminProtocol

Garmin (`www.garmin.com`) is one of the world's leading producers of GPS-equipment. Their range of products include handhelds and panel-mounted GPS-units for boats and cars. Common to all these products is the protocol[2] used to communicate with the GPS from other electronic devices, usually a PC.

Communication with this protocol takes place between two electronic devices: the GPS-unit and the host. The host is usually a PC, but could be any device implementing the protocol. This section is a short overview of how the Garmin-products communicate with the host. For a more detailed description of the protocol, refer to the protocol specification.

---

[2] http://www.garmin.com/support/commProtocol.html

The protocol is split into three layers:

Application
Link
Physical

2.3.1 Physical

The physical layer handles the transfer of data in byte-format between the two devices, and is based on RS-232 (PC serial cable). The serial-cable is running at a speed of 9600 baud. Data transfer is done full-duplex with 8 data bits, no parity and 1 stop bit. This is the default configuration that all Garmin-products support. Some products may be able to change these parameters to achieve higher performance.

2.3.2 Link

The link layer describes a packet format for transmitting data between the GPS and the host.
All packets have the following format:

byte:	Name:	Description:
0	DLE	Data Link Escape. Always 16 (10h).
1	ID	The type of packet.
2	Size	The size of the packet's data-field in bytes.
3 to n-4	Data	The packet's data-field.
n-3	Checksum	The 2's complement of byte 1 to n-4.
n-2	DLE	
n-1	ETX	End of text. Always 3.

If any byte, except byte 0 and n-2, has the value 16, it's immediately followed by another byte of the same value. This byte stuffing is not part of the checksum- and datasize-calculations.
Each packet, with a few exceptions, must be confirmed by the receiving device (ie. either the host or the GPS) with an acknowledgement - packet (ACK). If the packet was found to contain errors, a not-acknowledged-packet (NAK) is sent.

The link-layer is made up by three protocols: L000, L001 and L002:

L000 defines the basic ID-values used for preliminary communication and basic communication. Among these are the ACK and NAK-types and
types capable of asking the GPS about it's brand and which versions of the Application layer protocols it supports.

L001 is compatible with nearly of the Garmin-products. It defines packet-types for

transferring position, time, routes, waypoints and more.

L002 is used mostly by panel-mounted GPS-products (ie. for cars and boats).

2.3.3Application

The application-level protocol defines the transfer of long data-sets (eg. all the waypoints in the GPS) and the specific datatypes and fields of the packet's data.

3.Problemanalysisanddesignconsiderations

This section will describe some of the design-choices that were made during the production of the framework.

3.1GeographicUnitsforgeneralGPS -classes

The general GPS-classes (ie. the ones that doesn't depend on a specific GPS-type) uses latitude and longitude as a way of representing locations on the Earth's surface. This coordinate-system was picked because it is the most commonly used coordinate system today[3]. Classes exist for keeping the coordinates expressed in both radians and degrees.

3.2GPS -informationpropagation

When considering how a program using the framework should get information from the GPS, two methods were evaluated.

The first method is request-reply-communication: Asking the GPS for information would be like invoking a method, where the requested information would be returned immediately when the method was finished.
The primary advantage of this is simplicity. Querying a GPS information would simply be a matter of instantiating a GPS-class and calling the appropriate methods. It would also require less from the people implementing a GPS-protocol, since they wouldn't have to deal with multi-threading: the thread requesting would be responsible for both transmitting the request and receiving the answer.
After a brief consideration numerous disadvantages appeared: There would be no support for unrequested information. If the GPS had a mode for automatically transmitting information, the framework would not be able to take advantage of it. A program might get delayed unnecessarily while waiting for the reply from the GPS. Several classes using the same GPS might also cause problems and cause duplicate requests and answers to appear on the communication line.

The other method is using the Observer-design pattern[4]. With this method classes would register themselves with the GPS-class to receive information. No information would be returned immediately on request. Instead the information would be returned through a receive-method in the listener-class.
The advantages of this is that there will be no delay for the requesting thread. When the request is transmitted, the thread will be able to continue executing. Another advantage is that this method supports automatically transmitted information from

[3] http://math.rice.edu/~lanius/pres/map/mapcoo.html
[4] In Java this is more commonly referred to as "listener"-pattern.

the GPS. Having several classes listening to the same GPS would also be a trivial task.

The primary disadvantage of this method is complexity. The listener-class needs to add several methods, regardless of whether the information transmitted to these methods is of interest to the class. The complexity in implementing the GPS-class is also rising, since this needs functionality for registering listeners and distributing information.

The framework ended up using the Observer-method for communicating information received from the GPS. The primary reason for picking this method was that it allowed the framework to take advantage of automatic transmission of data from the GPS-unit.

3.3GPS -connection

How close should the library integrate with the javax.comm-package? The pros for close integration was that it would make the library even more accessible. The user would only have to specify the name of the communication port to use when initializing the GPS. This would make it very appealing and easy to use. The cons to this design is that it locks the library in a situation where it's only capable of communicating with GPS-units connected to the serial- and parallel-ports of the computer. This would prevent the library from adjusting easily to new ways of interfacing to devices, such as Bluetooth or Jini.
The library ended up trying to rely as little as possible on javax.comm. Instead it is up to the user to get Java-streams that connect to the GPS-unit and supply these to the GPS-implementation. Only the programs in the examples-package rely on javax.comm.

4.User'sGuide

4.1Prerequisites

Certain things are required for this library to work. First of all, a connection between the GPS and the host is needed. Usually this will be a cable connecting to the serial or (in rare cases) the parallel-port of the host.

A set of Java-classes able to facilitate the communication method is also needed. In the case of serial-communication, the Java Communications API[5] is recommended. This is also required in order to run the examples bundled with the GPS-library.

[5] http://java.sun.com/products/javacomm/

4.2 Package structure

The framework-package is made with the following structure:

`dk.itu.haas.GPS`

Contains the generic GPS-classes and interfaces needed to listen to a GPS-unit.

`dk.itu.haas.GPS.Garmin`

Contains the Garmin-implementation and Garmin-specific classes. Unless you want to makes changes to the implementation, the only class needed from this package is `GarminGPS`.

`dk.itu.haas.GPS.services`

This package contains the services developed using the framework. In this version, only the `AreaAlarm`-service is in this package.

`dk.itu.haas.GPS.examples`

This package currently contains two examples: `ConnectionTest` and `AreaAlarmDemo`.
ConnectionTest can be used to verify if you have a working connection to the GPS. It will open a connection and ask the GPS for a description, which will be printed to the console.
See the *"Using the AreaAlarm-service"* - section for a description of the `AreaAlarmDemo`-program.

The `javadoc`-directory contains the documentation of the framework in the javadoc format, which should be easily recognizable by all Java-developers. This report is only meant to give an overview of the framework and the design process, so consult the `javadoc`-directory for the most in-depth descriptions of a specific class or method.

4.3 Using the GPS-library with a Garmin-GPS

Make sure that `gps.jar` is in your classpath. Verify that the connection between the GPS-unit and the host is working.
An easy way to do this is by running the `ConnectionTest` – program (in the examples-package), which will connect to the GPS and query it for a description of its services. Be sure to configure the connection correctly. Some GPS-units are very strict with regards to data-speed and other settings.

When the basic connection is ready, it's possible to start working with the library. The most important class is `dk.itu.haas.GPS.GPS`. This class encapsulates all the functionality that the generic GPS-classes require from the GPS-unit.

The first thing to do is open the streams needed to communicate with the GPS, ie. an input- and an output-stream.
Next step is to instantiate the GPS-class corresponding to the GPS-unit. The constructor of the GPS will take the streams as arguments.

The GPS is now ready for transmitting data. The way to receive data from the GPS is by registering a class as a listener with the GPS-class.

There are two kind of listeners: GPS-listeners and Waypoint-listeners. In order to become a listener, a class must implement the corresponding interface.
A GPS-listener-class must implement the interface `IGPSlistener`. This will add the following methods:

```
public void timeReceived(ITime t);

public void dateReceived(IDate d);

public void positionReceived(IPosition pos);
```

These methods will be called whenever the GPS transmits information about the current time, the current date or the current position respectively. `ITime`, `IDate` and `IPosition` are interfaces that contains method for reading the appropriate information. Consult the documentation for more information on how to use them.

Remember that `ITime`, `IDate` and `IPosition` are only interfaces. The references being sent to these methods might not always point to instances of the same class. It's risky to try to cast the reference to its original type!

The other availabe listener-type is the waypoint-listener. This listener allows a class to receive the waypoints stored in the GPS-unit. To become a waypoint-listener, a class must implement the `IWaypointListener`-interface. This will add the following methods:

```
public void transferStarted(int number);

public void waypointReceived(IWaypoint wp);

public void transferComplete();
```

`transferStarted` will be called when the GPS starts transferring waypoints. `number` is the number of waypoints in the transmission. `waypointReceived` is called for each waypoint. `transferComplete` is called after the waypoints has been transmitted.

The listener-methods are called by a thread responsible for dispatching the received data to the listeners. Be sure not to tie up this thread too much, ie. don't perform too many tasks in these methods. A good way of avoiding that is to use other threads to perform the heavy tasks. Use the `Object.wait` - method to let them sleep until information from the GPS arrives. Let the dispatcher-thread call `Object.notify` when it delivers the information.

Example:
```
ITime time = null;
Object timewait = new Object();
public void timeReceived(ITime t) {
  time = t;
  timewait.notify();
}

// The run-method of a Thread-object.
public void run() {
  while (true) {
    if (time == null)
      timewait.wait();//Notice: Exception-catching removed.
      print(time); // Perform big operations.
      time = null;
    }
}
```

Information can be dispatched in two ways: automatic or by request. Use the request-methods to order the GPS to transmit data. The automatic mode can be entered by using the `setAutoTransmit`-method. When setting automatic mode, the GPS will transmit time, data and position about once per second.

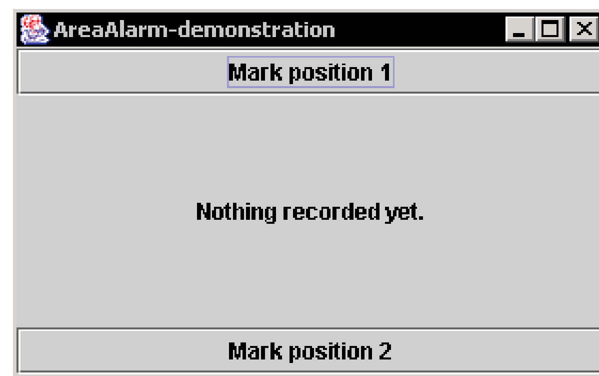
To stop listening to the GPS, use the `shutdown`-method. This will cause the dispatching-thread to stop. If the method is called with *`true`*, the GPS will be asked to turn off.

4.4 Using the AreaAlarm -service

The arealarm-service is a small location-based service made with the framework. It lets classes register for notification whenever the GPS-unit enters or exits a rectangular area. The rectangular area is defined by two points, which each mark two opposite corners of the rectangle. To become a listener of the `AreaAlarm`-service, a class must implement the IAreaAlarm-interface. It defines two methods:

> `public void exitedAlarm();`
> This method is called when the GPS exits the area.
> `public void enteredAlarm();`
> This method is called when the GPS enters the area.

A simple example-program (`dk.itu.haas.GPS.examples.AreaAlarmDemo`) using the AreaAlarm-service has been developed. Running the program looks like this:



Press the buttons on the top and the bottom of the window to mark the 2 points that will define the rectangle. When both points have been marked, the text in the middle of the window will display whether the GPS is inside or outside the defined area. Be careful not to mark the same position twice! Also make sure that the rectangle has an area, ie. that both latitude and longitude in the two positions differ.

5.Developer'sGuide

5.1Extendingdk.itu.haas.GPS.GPS

While creating this library, efforts has been made to make it extendable. There are two ways of extending it: By making it support another GPS-protocol or by adding functionality to the core classes. Adding support for other GPS-protocols is covered in this section. Refer to the next chapter (implementation details) for information on the architecture of the library. Read the conclusion-chapter for suggestions on how to improve the library.

Adding support for a new GPS-protocol requires making a new class that extends `dk.itu.haas.GPS.GPS`. Usually a GPS-implementation will ask for an input- and an output-stream in its constructor. This will allow the person using it to freely select the communication method. It might, for example, be possible to use Bluetooth or a similar wireless protocol to communicate with future GPS-units.

The new GPS-class needs to have a combined listening and dispatching thread. It's very important that this thread does not lock while waiting for data from the GPS. This can be achieved by wrapping the inputstream in a `BufferedInputStream`.

Instead of just calling the `read`-method on the inputstream, call `available` to check if anything is ready to be read. If not, sleep for some time, and try again.

```
while (active) {
  if (input.available() == 0) {
    Thread.sleep(500); // Exception catching removed.
    continue;
  }
}
```

The advantage of reading from the GPS like this, is that the thread will constantly check whether it's supposed to continue or whether it has been asked to shut down. Calling `read` when nothing is coming from the GPS will result in the thread being locked indefinately, which in turn may cause the whole program to lock when it's supposed to shut down.

The other task of the thread is to propagate the received information to the listeners. This task is accomplished by encapsulating the information in classes which implement the corresponding interface, ie. if you have data containing position, implement the `IPosition`-interface. To propagate information to the listeners, call one the following methods:

```
void firePositionData(IPosition pos)
void fireDateData(IDate dat)
void fireTimeData(ITime time)

void fireTransferStart(int number)
void fireWaypointData(IWaypoint wp)
void fireTransferComplete();
```

These methods are all implemented in the `dk.itu.haas.GPS.GPS`-class, together with methods for handling registration of listeners, so the GPS-implementation does not need worry about this.

For the sake of debugging or the curious user, it's also a good idea to implement a listener for listening directly to the data coming in from the GPS. In the case of the Garmin-protocol, this is implemented by forwarding all received packets directly to the listener.

In the following is a list of the methods that needs to be implemented, and their intended behaviour, when making a new GPS-class. Remember that all the following methods, except `getDescription`, will not return a result directly, but instead make a request to the GPS. The answer from the GPS should then be forwarded to the requesting class through the listeners.

`public void requestPosition()`
Ask the GPS to send it's current position.

`public void requestTime()`
Ask the GPS to send the current time.

`public void requestDate()`
Make the GPS return the current date.

`public String getDescription()`
Get a human-readable string describing the GPS and its capabilities. Notice the difference between this method and the other request-methods. Since the result of calling this method will be static all through the use of the GPS, there is no reason to make listeners for the result, and the description is therefore returned immediately. It may be necessary to collect this information and save it as soon as the GPS-class is instantiated.

`public void requestWaypoints()`
Ask the GPS to transmit all of its waypoints.

`public void setAutoTransmit(boolean t)`
If `t` is *`true`*, the GPS should start automatically transmitting position- and time-data periodically (about once per second is a good frequency). If the GPS does not support a mode like this, the programmer can either choose to throw a `FeatureNotSupportedException` or implement a thread that constantly queries the GPS for the data. If `t` is *`false`*, this behaviour will be turned off.

`public void shutdown(boolean b)`
Calling this method will ask the listener-thread to stop listening on the GPS. If `b` is *`true`*, the GPS should physically turn off when this method is called. Throwing a `FeatureNotSupportedException`, if this is not possible, is optional.

In all of the above methods, it's possible to throw a `FeatureNotSupportedException` if the GPS is not capable of doing what the method requests.

This was a description on how to implement the external interface of the GPS-class. Look to the section describing the implementation of the Garmin-GPS for inspiration on how to handle things internally in the class.

6.Implementationdetails

6.1GeneralGPS -classes

The classes and interfaces making up the "general GPS-classes" are located in the `dk.itu.haas.GPS-package`. These are the generic GPS-classes that are used regardless of which GPS-unit the program is connected to.

The most important class is GPS. This class is an abstract base-class that all GPS-implementations must inherit from. It contains request-methods that must be implemented in the base-classes and methods for registering listeners and propagating information from the GPS-implementation to the listeners.
Beyond the GPS-class, this package contains mostly interfaces, which can be divided into two groups: data-interfaces and listener-interfaces.

The listener-interfaces are implemented by a class that wishes to listen for information from the GPS. The methods defined in these interfaces use the data-interfaces as arguments. The data-interfaces are implemented by classes encapsulating the data from a GPS. This makes it possible to propagate information from any GPS-type, through the same channels, to the same listeners.

The data–interfaces does not contain methods for changing or manipulating the data. This is prevented because the reference to the data may be sent out to several listeners, so it would be inconsistent to allow one of them to change it. If a class needs to change a position, it should use the `Position`-class which will make a new set of the data. If a class needs to hold on to a reference to a position for a long time, it's also a good idea to convert it to the `Position`-class. The reason is that it's not possible to tell how much data is used in the object referenced by the data-interface. It's likely to be the entire packet in byte-form together with the relevant data. Keeping a reference to this object will keep a lot of unnecessary data from being garbage collected.

6.2Garmin -implementation

6.2.1Basiccommunication

The basic input and output to the GPS is handled by two special stream-classes: GarminInputStream and GarminOutputStream. The input-class strips the input of the byte-stuffing (ie. the double DLE-bytes. See the section describing the Garmin-protocol), while the output-class makes sure to add byte-stuffing to all outgoing traffic. Both streams have added methods that allows them to respectively send and receive entire packets.

6.2.2Th eGarminPacket -class

The `readPacket`-method in the `GarminInputStream` returns a byte-array. To encapsulate the information contained in that array, the `GarminPacket`-class was created. This class checks the integrity of the received packet, ie. the checksum and the length of the data-field. It throws runtime-exceptions in case of malformed packets. The class also contains methods for converting datatypes from the byte-form of the packet to the equivalent Java-datatype.

The class also contains factory-methods for creating some common packets. The `createCommandPacket`-method will return a Garmin-packet that can be transmitted as a request to the GPS. The constructor `GarminPacket(int[], boolean)` is used in the case of packets being created from scratch (ie. not received from the GPS). The boolean-argument will determine whether the class will calculate and insert the checksum of the packet.

The method `CreateBasicPacket(int t, int[] d)` will create a packet of type `t` and put the array `d` in the data-field of the packet. All the relevant packet-types exist as constant integer-types in the `GarminPacket`-class.

Several classes have been created that extends `GarminPacket`. Each new class specialises in handling the information contained in a specific kind of packet. The constructors of these packets makes sure to initially validate the packet through the GarminPacket-constructor and next validate the specific kind of packet. After the validation, the relevant data is extracted from its byte-form in the datafield of the packet. Some specialisations of `GarminPacket` implements one or more of the interfaces of the general GPS-implementation, and can thus be propagated to listeners of the GPS-class.
The following classes extend the `GarminPacket`:

`PositionDataPacket`. Interface: `IPosition`
Packet containing information about the current position of the GPS.

`TimeDataPacket`. Interfaces: `ITime` and `IDate`
Packet containing information about the current time and date.

`RecordsPacket`. No interfaces implemented.
This type of packet contains information about an upcoming multi-packet transfer of data.

`PVTDataPacket`. Interfaces: `IPosition` and `ITime`
The automatic transmission mode of the Garmin-protocols sends this type of packet, which contains information about current position and time.

ProtocolDataPacket. No interfaces implemented.
This packet is sent by the GPS as a notification of which datatypes and protocol-versions it supports.

`ProductDataPacket`. No interfaces implemented.
This packet contains information about the type of the GPS and the version of the software in it.

`WaypointDataPacket`. Interfaces: `IWaypoint`.
A packet containing a waypoint.

The `WaypointDataPacket` stands out from the rest, in that the Garmin protocol defines 14 different formats of the packet. Which format is used by a specific GPS-unit can be seen from its `ProtocolDataPacket`. The static field `datatypeversion` in the `WaypointDataPacket` contains the value which is used together with the GPS. In this version of the library, only one version (D108) of the waypoint-packet is implemented. The reason is, that D108 was the only version supported by the GPS - unit used while making the library, and thus it would be impossible to test an implementation of the other versions.

6.2.3 The dispatching thread

The `GarminGPS`-class contains a dispatcher thread. Its task is to receive packets from the GPS and distribute them to the listeners. The distribution itself is done by methods in the `GPS`-class, while the dispatcher thread determines which kind of information the packet contains and to which listeners it should be sent.

6.2.4 The GarminListener -interface

The `GarminListener`-interface enables the curious user to receive references to all the `GarminPacket`s transmitted to the host This can be useful for several things. The user might want to extend the Garmin-package, and therefore needs to monitor the communication, or it can be used to debug the framework with other Garmin-units than the one it was written with. It's recommended that all GPS-implementations include a similar interface.

7.Testing

The lack of a notebook has prevented field-tests of the library. Testing the library has been done through the demo-mode of the GPS-unit. Data parsed from the packets has been compared to the equivalent data on the GPS-unit's display. This method has revealed that no obvious errors exist in the library.

8.Conclusion

The result of the project meets the criteria set up at the beginning. The result is a Java-library capable of adding GPS-functionality to other programs. The library is only able to communicate with a Garmin-type GPS, but should be easily extendable.

The library has a few shortcomings. Worst of all, the time returned from the Garmin-GPS will always be in Greenwich mean time. It has not been possible to get the GPS to supply an offset to get the local time.
The Garmin-GPS-units support a lot of formats for the waypoint-datatype. Unfortunately the GPS that was used to develop this library only supports one of these, and therefore only that one is implemented in the library.

An Area-alarm-service was developed using the library. This will allow a class to receive notification whenever the GPS-unit enters or exits an area.

The library could be improved in several ways. First and foremost, more GPS-types could be added, making it compatible with more communication-protocols. A standard GPS today supports a rather large variety of functions (routes, tracks etc.). An obvious task would be to add support for these functions to the library.

Smaller scale improvements include design issues that didn't make it to this release. At the moment a class can listen to at most one AreaAlarm. The reason is that the AreaAlarm-object does not identify itself when propagating an event, and thus a listener registered with several AreaAlarms will not know from which the event comes. An improvement of this service would be to implement that the AreaAlarm identifies itself when firing an event.

Some classes might benefit from being notified whenever something happens to the state of the GPS, eg. the auto-transmisson mode is enabled or the GPS is turned off. A listener being able to receive notifications of this kind of events might be relevant.

It would be a good feature to distinguish between automatically transmitted data and requested data. As several classes can listen to the same GPS, it's not optimal that one of them can turn on the auto-transmission feature and thus flood the other with information. Or even worse, one listener can turn off auto-transmission, which

other classes depend on. A good way to work around this could be to have auto-transmisson turned on by default, and add a seperate listener-interface for automatically transmitted data.

Henrik Aasted Sørensen, 2002

9.References

The Garmin Protocol Specification 001-00063-00 Rev. 3
- http://www.garmin.com/support/commProtocol.html

Trimble – All about GPS.
- http://www.trimble.com/gps/

10.Appendix

10.1dk.itu.haas.GPS –package

10.1.1GPS

```
package dk.itu.haas.GPS;
import java.util.Vector;

/**
* This is the abstract base-class that encapsulates the functionality of a generic GPS-unit.
*/
public abstract class GPS {
	/** A vector containing references to the objects registered as listeners on this GPS. */
	protected Vector GPSlisteners;

	/** A vector containing references to the objects registered as listeners for waypoint-data from
	the GPS.*/
	protected Vector WaypointListeners;

	/** A vector containing references to the objects registered as listeners for transfers from the
	GPS.
	* A listener can't be directly add to this group, but will instead be added when it registers for
	some
	* other data that is being transmitted in transfer-groups.
	*/
	protected Vector TransferListeners;

	protected GPS() {
		GPSlisteners = new Vector();
		WaypointListeners = new Vector();
		TransferListeners = new Vector();
	}

	/** Adds the specified IGPSlistener to receive data from the GPS. */
	public void addGPSlistener(IGPSlistener l) {
		// Only allow a listener to be registered once.
		if (GPSlisteners.contains(l))
			return;

		GPSlisteners.add(l);
		return;
	}
	/**
	* Adds l to the list of listeners interested in transfer-events.
	* Members of this list can't be directly added, but have to be added through addition of
	* other listeners.
	*/
	protected void addTransferListener(ITransferListener l) {
		// Only allow a listener to be registered once.
		if (TransferListeners.contains(l))
			return;

		TransferListeners.add(l);
		return;
	}

	/**
	* Adds l to the list of listeners interested in waypoint-data.
	* Also adds l to the list of transfer-listeners.
	*/
	public void addWaypointListener(IWaypointListener l) {
		// Only allow a listener to be registered once.

		if (WaypointListeners.contains(l))
			return;

		addTransferListener(l);

		WaypointListeners.add(l);
		return;
	}
```

```
/**
* Removes the the Waypoint-listener l from the list of Waypoint-listeners.
*/
public void removeWaypointListener(IWaypointListener l) {
	while (WaypointListeners.removeElement(l)) {}
	return;
}

/**
* Removes the the GPS-listener l from the list of GPS-listeners.
*/
public void removeGPSListener(IGPSlistener l) {
	while (GPSlisteners.removeElement(l)) {}
	return;
}

/**
* Removes the the transfer-listener l from the list of transfer-listeners.
*/
protected void removeTransferListener(ITransferListener l) {
	while (TransferListeners.removeElement(l)) {}
	return;
}

/**
* Notifies listeners of the beginning of a stream of data. Tells listeners of the number of
* data-units in the transfer.
*/
public void fireTransferStart(int number) {
	for (int i = 0 ; i < TransferListeners.size() ; i++) {
		((ITransferListener) TransferListeners.elementAt(i)).transferStarted(number);
	}
}

/**
* Goes through the list of Waypoint-listeners and distributes the waypoint wp.
*/
public void fireWaypointData(IWaypoint wp) {
	for (int i = 0 ; i < WaypointListeners.size() ; i++) {
		((IWaypointListener) WaypointListeners.elementAt(i)).waypointReceived(wp);
	}
}

/**
* Notifies listeners of the end of a stream of data.
*/
public void fireTransferComplete() {
	for (int i = 0 ; i < TransferListeners.size() ; i++) {
		((ITransferListener) TransferListeners.elementAt(i)).transferComplete();
	}
}

/**
* Goes through the list of GPSlisteners and distributes the new position data.
*/
protected void firePositionData(IPosition pos) {
	for (int i = 0 ; i < GPSlisteners.size() ; i++) {
		((IGPSlistener) GPSlisteners.elementAt(i)).positionReceived(pos);
	}
}

/**
* Goes through the list of GPSlisteners and distributes the new date data.
*/
protected void fireDateData(IDate dat) {
	for (int i = 0 ; i < GPSlisteners.size() ; i++) {
		((IGPSlistener) GPSlisteners.elementAt(i)).dateReceived(dat);
	}
}

/**
* Goes through the list of GPSlisteners and distributes the new time data.
*/
protected void fireTimeData(ITime time) {
	for (int i = 0 ; i < GPSlisteners.size() ; i++) {
		((IGPSlistener) GPSlisteners.elementAt(i)).timeReceived(time);
	}
```

```
    }

    /** Makes a request for the specified data to the GPS. Data will be returned to all listeners
    through the IGPSlistener-interface. */
    public abstract void requestPosition();

    /** Makes a request for the specified data to the GPS. Data will be returned to all listeners
    through the IGPSlistener-interface. */
    public abstract void requestTime();

    /** Makes a request for the specified data to the GPS. Data will be returned to all listeners
    through the IGPSlistener-interface. */
    public abstract void requestDate();

    /**
    * Requests a descriptive string from the GPS. Should be formatted for human-reading.
    * The string should be constructed by every GPS-implementation upon startup.
    */
    public abstract String getDescription();

    /**
    * Asks the GPS to transmit all the waypoints in it's memory. The result will be returned through
    the WaypointListener-interface.
    * Throws a FeatureNotSupportException if it isn't possible to do this on the GPS.
    */
    public abstract void requestWaypoints();

    /**
    * Asks the GPS to either start or stop transmitting data periodically. <br/>
    * The data will be the that which is accessible through the IGPSlistener-interface.
    * Throws a FeatureNotSupportException if it isn't possible to do this on the GPS.
    */
    public abstract void setAutoTransmit(boolean t);

    /** Stops communication with GPS.<br/>
    * Most likely, your program won't be able to shut down before you've called this method.
    * If b is set to true, the GPS will be asked to turn off.
    * If b is set to false, the GPS will remain turned on.
    * Throws a FeatureNotSupportException if b is true, but the GPS-unit doesn't support that function.
    */
    public abstract void shutdown(boolean b);
}
```

10.1.2 IGPSlistener

```
package dk.itu.haas.GPS;

/**
* This interface is used to receive notification each time the GPS transmits one of the common data, ie.
position, time and date.
* <ul>
* <li> The GPS does not necessarily transmit these things periodially by itself! Some GPS-units needs a
request before
* transmitting anything. Use the method GPS.setAutoTransmission(true) if you want the GPS to periodically
send this data.
* <li> Don't perform any long calculations or big operations in these methods. They're called by a
dispatching thread, and putting
* it to too much work will slow performance on the communication with the GPS.
* </ul>
*/

public interface IGPSlistener {
  /** Invoked when the GPS transmits time-data. */
  public void timeReceived(ITime t);
  /** Invoked when the GPS transmits date-data. */
  public void dateReceived(IDate d);
  /** Invoked when the GPS transmits position-data. */
  public void positionReceived(IPosition pos);
}
```

10.1.3IwaypointListener

```
package dk.itu.haas.GPS;

public interface IWaypointListener extends ITransferListener {
  /**
  * This method is called whenever a waypoint is received from the GPS.
  */
  public void waypointReceived(IWaypoint wp);
}
```

10.1.4ItransferListener

```
package dk.itu.haas.GPS;

/**
* The methods in this interface are used whenever the GPS should transfer a series of data.
*/
public interface ITransferListener {
  /**
  * This method is called when a transfer is initiated. <br/>
  * Number is the amount of data that will be transferred, ie. the amount of waypoints.
  * If it's not possible to tell how much data that will be transferred, number will be -1.
  */
  public void transferStarted(int number);

  /**
  * This method is called when the transfer is complete.
  */
  public void transferComplete();
}
```

10.1.5IPosition

```
package dk.itu.haas.GPS;

/**
* This interface is implemented by all packets capable of returning a position.
*/
public interface IPosition {
  /**
  * This method returns the latitude of the position.
  */
  public PositionRadians getLatitude();

  /**
  * This method returns the longitude of the position.
  */
  public PositionRadians getLongitude();
};
```

10.1.6ITime

```
package dk.itu.haas.GPS;

/**
* This interface is implemented by all packets capable of returning the time of day.
*/
public interface ITime {
  /** Returns the hour of the day. */
  public int getHours();

  /** Returns the minute of the hour. */
  public short getMinutes();

  /** Returns the second of the minute. */
  public short getSeconds();
};
```

10.1.7IDate

```
package dk.itu.haas.GPS;

/**
* This interface is implemented by all packets capable of returning a date.
*/
public interface IDate {
  /** Returns the day of the month. */
  public short getDay();

  /** Returns the month. */
  public short getMonth();

  /** returns the year. */
  public int getYear();
};
```

10.1.8IWaypoint

```
package dk.itu.haas.GPS;

public interface IWaypoint extends IPosition {
  public String getName();
}
```

10.1.9Position

```
package dk.itu.haas.GPS;

/**
* This is a class meant for containing positions.
*/

public class Position implements IPosition {
  private PositionRadians lat, lon;

  /**
  * Makes a new position. Initializes the latitude and the longitude to 0.
  */
  public Position() {
          this(0,0);
  }

  /**
  * Initializes the Position with la as the latitude and lo as the longitude.
  */
  public Position(double la, double lo) {
          lat = new PositionRadians(la);
          lon = new PositionRadians(lo);
  }

  /**
  * Initializes the position object from an IPosition reference.
  */
  public Position(IPosition pos) {
          lat = pos.getLatitude();
          lon = pos.getLongitude();
  }

  /**
  * Sets the latitude of this position.
  */
  public void setLatitude(PositionRadians l) {
          lat = l;
  }

  /**
  * Sets the longitude of this position.
  */
  public void setLongitude(PositionRadians l) {
          lon = l;
  }
```

```
  /**
  * Returns the latitude of this position.
  */
  public PositionRadians getLatitude() {
          return lat;
  }

  /**
  * Returns the longitude of this position.
  */
  public PositionRadians getLongitude() {
          return lon;
  }
}
```

10.1.10PositionDegrees

```
package dk.itu.haas.GPS;

/**
* Class used to store degrees, usually latitude or longitude.
*/

public class PositionDegrees {
  protected double value;

  public PositionDegrees(double v) {
          value = v;
  }

  /**
  * Returns the degrees part of this object, when converted to coordinates.
  */
  public int getDegrees() {
          return (int) value;
  }

  /**
  * Converts the degrees to Radians.
  */
  public PositionRadians convertToRadians() {
          return new PositionRadians(( value * Math.PI ) / 180.0d);
  }

  /**
  * Returns the minutes part of this object, when converted to coordinates.
  */
  public double getMinutes() {
          double v = value;
          v -= getDegrees();
          return 60 * v; // 60 minutes in one degree.
  }
}
```

10.1.11PositionRadians

```
package dk.itu.haas.GPS;

/**
* Class used to store radians, usually latitude or longitude.
* Contains methods for converting to the format degress,minutes.
*/
public class PositionRadians {
  private double value;

  /**
  * Initializes the PositionRadians-object. After the object is constructed, it can't change is value.
  */
  public PositionRadians(double v) {
          value = v;
  }

  public double getRadians() {
          return value;
```

```
    }

    /**
    * Returns the degrees part of this object, when converted to coordinates.
    */
    public int getDegrees() {
            return (int) (value * (180.0d / Math.PI));
    }

    /**
    * Returns the minutes part of this object, when converted to coordinates.
    */
    public double getMinutes() {
            double v = value * (180.0d / Math.PI);
            v -= getDegrees();
            return 60 * v; // 60 minutes in one degree.
    }

    /**
    * Returns the value of this object in degrees and minutes.
    */
    public String toString() {
            return getDegrees() + ":" + getMinutes() + "";
    }

    /**
    * Tests if the two PositionRadians contains the same value.
    */
    public boolean equals(PositionRadians p) {
            if (value == p.value)
                    return true;
            else
                    return false;
    }

    /**
    * Tests if this PositionRadians is greater than p.
    */
    public boolean greaterThan(PositionRadians p) {
            if (value > p.value)
                    return true;
            else
                    return false;
    }

    /**
    * Tests if this PositionRadians is smaller than p.
    */

    public boolean smallerThan(PositionRadians p) {
            if (value < p.value)
                    return true;
            else
                    return false;
    }
}
```

10.1.12 FeatureNotSupportedException

```
package dk.itu.haas.GPS;

/**
* Thrown by methods in classes extending the GPS-class, if the implemented GPS-unit does not support the
feature
* requested in the method.
*/
public class FeatureNotSupportedException extends RuntimeException {
}
```

10.2 dk.itu.haas.GPS.Garmin –package

10.2.1 GarminGPS

```
package dk.itu.haas.GPS.Garmin;
import dk.itu.haas.GPS.*;
import java.io.*;
import java.util.Vector;

public class GarminGPS extends GPS implements Runnable {
  GarminInputStream input;
  GarminOutputStream output;

  /** A human-readable description of the GPS-unit. */
  protected String description;

  Thread listener;
  /** The listening thread will be active as long as this variable remains true. */
  protected boolean active;
  /** A vector containing references to all the GarminListeners. */
  protected Vector GarminListeners;

  /** */
  public GarminGPS(BufferedInputStream i, BufferedOutputStream o) {
          input = new GarminInputStream(i);
          output = new GarminOutputStream(o);
          listener = new Thread(this);
          listener.start();
          active = true;
          GarminListeners = new Vector();

          // Request product information.
          try {
            output.write(GarminPacket.createBasicPacket(GarminPacket.Pid_Product_Rqst, new int[] {}));
          } catch(IOException e) {}
  }


  /**
  * Adds the specified GarminListener to receive all packets sent from the GPS.
  */
  public void addGarminListener(GarminListener l) {
          // Only allow a listener to be registered once.
          if (GarminListeners.contains(l))
                  return;

          GarminListeners.add(l);
          return;
  }

  /**
  * Removes the specified GarminListener from the list of listeners.
  */
  public void removeGarminListener(GarminListener l) {
          while (GarminListeners.removeElement(l)) {}
          return;
  }

  /**
  * Goes through the list of GarminListeners and transmits p to them.
  */
  protected void fireGarminPacket(GarminPacket p) {
          for (int i = 0 ; i < GarminListeners.size() ; i++) {
                  ((GarminListener) GarminListeners.elementAt(i)).GarminPacketReceived(p);
          }
  }

  /** This method is listening for input from the GPS. */
  public void run() {
          GarminPacket pack = null;

          while (active) {
                  try {
                          if (input.available() == 0) {
```

```
                                try {
                                        Thread.sleep(500);
                                } catch(InterruptedException e) {}
                                continue;
                        }
                        pack = new GarminPacket(input.readPacket());
                } catch (IOException e) {
                        active = false;
                        return;
                } catch (InvalidPacketException e) {
                        // Send back a NAK-packet.
                        try {
                          output.write( GarminPacket.createBasicPacket(GarminPacket.Pid_Nak_Byte, new
int[] {pack.getID(), 0}));
                        } catch (IOException ex) {
                                active = false;
                                return;
                        }
                }

                // Send back ACK-packet.
                try {
                  output.write( GarminPacket.createBasicPacket(GarminPacket.Pid_Ack_Byte, new int[]
{pack.getID(), 0}));
                } catch (IOException e) {
                        active = false;
                }

                fireGarminPacket(pack);
                Distribute(pack);

        } // End of while
  }

  /** This method is used to identify the type of packet received, and distribute it to the correct
  * listeners.
  */
  protected void Distribute(GarminPacket p) {
        switch (p.getID()) {
                case GarminPacket.Pid_Position_Data :
                        firePositionData(new PositionDataPacket(p));
                        return;
                case GarminPacket.Pid_Date_Time_Data :
                        TimeDataPacket tdp = new TimeDataPacket(p);
                        fireDateData(tdp);
                        fireTimeData(tdp);
                        return;
                case GarminPacket.Pid_Pvt_Data :
                        PVTDataPacket pvtp = new PVTDataPacket(p);
                        fireTimeData(pvtp);
                        firePositionData(pvtp);
                        return;
                case GarminPacket.Pid_Records :
                        fireTransferStart((new RecordsPacket(p)).getNumber());
                        return;
                case GarminPacket.Pid_Wpt_Data :
                        fireWaypointData( new WaypointDataPacket(p));
                        return;
                case GarminPacket.Pid_Xfer_Cmplt :
                        fireTransferComplete();
                        return;
                case GarminPacket.Pid_Product_Data :
                        System.out.println("Product data arrived!");
                        ProductDataPacket pp = new ProductDataPacket(p);
                        description = pp.getDescription();
                        description += "\nSoftware version: " + pp.getSWVersion();
                        description += "\nProduct ID: " + pp.getProductID();
                        return;
                case GarminPacket.Pid_Protocol_Array :
                        description += "\nProtocols supported:\n";
                        description += (new ProtocolDataPacket(p)).toString();
                        return;
                default :
                        return;
        }
  }
```

```
  /** Makes a request for the specified data to the GPS. Data will be returned to all listeners through
the IGPSlistener-interface. */
  public void requestPosition() {
          try {
                  output.write( GarminPacket.createCommandPacket(GarminPacket.Cmnd_Transfer_Posn));
          } catch (IOException e) {}
  }
  /** Makes a request for the specified data to the GPS. Data will be returned to all listeners through
the IGPSlistener-interface. */
  public void requestTime() {
          try {
                  output.write( GarminPacket.createCommandPacket(GarminPacket.Cmnd_Transfer_Time));
          } catch (IOException e) {}
  }

  /** Makes a request for the specified data to the GPS. Data will be returned to all listeners through
the IGPSlistener-interface. */
  public void requestDate() {
          try {
                  output.write( GarminPacket.createCommandPacket(GarminPacket.Cmnd_Transfer_Time));
          } catch (IOException e) {}
  }

  /**
  * Asks the GPS to transmit all the waypoints in it's memory. The result will be returned through the
WaypointListener-interface.
  */
  public void requestWaypoints() {
          try {
                  output.write( GarminPacket.createCommandPacket(GarminPacket.Cmnd_Transfer_Wpt));
          } catch (IOException e) {}
  }

  /**
  * Asks the GPS to either start or stop transmitting data periodically. <br/>
  * The data will be that which is accessible through the IGPSlistener-interface.
  * Throws a FeatureNotSupportException if it isn't possible to do this on the GPS.
  */
  public void setAutoTransmit(boolean t) {
          if (t) {
                  try {
                  output.write( GarminPacket.createCommandPacket(GarminPacket.Cmnd_Start_Pvt_Data));
                  } catch (IOException e) {}
          } else {
                  try {
                          output.write(
GarminPacket.createCommandPacket(GarminPacket.Cmnd_Stop_Pvt_Data));
                  } catch (IOException e) {}
          }
  }

  /** Stops communication with GPS.<br/>
  * Most likely, your program won't be able to shut down before you've called this method.
  * If b is set to true, the GPS will be asked to turn off.
  * If b is set to false, the GPS will remain turned on.
  */
  public void shutdown(boolean b) {
          if (b) {
                  try {
                          output.write( GarminPacket.createCommandPacket(GarminPacket.Cmnd_Turn_Off_Pwr));
                  } catch (IOException e) {}
          }
          active = false;
  }

  /**
  * Returns a string telling the brand of the Garmin-gps, software version and the protocols supported.
  */
  public String getDescription() {
          return description;
  }
}
```

10.2.2 GarminPacket

```
package dk.itu.haas.GPS.Garmin;
import dk.itu.haas.GPS.*;
/**
* A class that encapsulates the basic functionality of a packet.
*/
public class GarminPacket {
  // L000 values. (Names taken from protocol specification.)
  public static final int Pid_Ack_Byte = 6;
  public static final int Pid_Nak_Byte = 21;
  public static final int Pid_Protocol_Array = 253;
  public static final int Pid_Product_Rqst = 254;
  public static final int Pid_Product_Data = 255;

  // L001 values.
  public static final int Pid_Command_Data = 10;
  public static final int Pid_Xfer_Cmplt = 12;
  public static final int Pid_Date_Time_Data = 14;
  public static final int Pid_Position_Data = 17;
  public static final int Pid_Records = 27;
  public static final int Pid_Wpt_Data = 35;
  public static final int Pid_Pvt_Data = 51;


  // A010 - Device Command Protocol.
  /** Abort current transfer. */
  public static final int Cmnd_Abort_Transfer = 0;
  /** Transfer almanac. */
  public static final int Cmnd_Transfer_Alm    = 1;
  /** Transfer position. */
  public static final int Cmnd_Transfer_Posn   = 2;
  /** Transfer proximity waypoints. */
  public static final int Cmnd_Transfer_Prx    = 3;
  /** Transfer routes. */
  public static final int Cmnd_Transfer_Rte    = 4;
  /** Transfer time. */
  public static final int Cmnd_Transfer_Time   = 5;
  /** Transfer track log. */
  public static final int Cmnd_Transfer_Trk    = 6;
  /** Transfer waypoints. */
  public static final int Cmnd_Transfer_Wpt    = 7;
  /** Turn off power. */
  public static final int Cmnd_Turn_Off_Pwr    = 8;
  /** Start transmitting PVT (Position, velocity, time) Data. */
  public static final int Cmnd_Start_Pvt_Data = 49;
  /** Stop transmitting PVT (Position, velocity, time) Data. */
  public static final int Cmnd_Stop_Pvt_Data   = 50;

  // Packet Boundaries.
  /**
  * Data link escape. Packet boundary.
  */
  public static final int DLE = 16;
  /**
  * End of text. Packet boundary.
  */
  public static final int ETX = 3;

  /**
  * The packet in byte-form.
  * It is required that the array-length is trimmed to the size of the packet.
  */
  protected int[] packet;

  /**
  * Creates a new GarminPacket with the contents of p.
  * Throws InvalidPacketException if packet is malformed.
  */
  public GarminPacket(int[] p){
         this(p, false);
  }

  /**
  * Creates a new GarminPacket with the contents of p.
  * if calcChecksum is true, the packet will have it's checksum recalculated.
```

```
* Throws InvalidPacketException if packet is malformed.
*/

public GarminPacket(int[] p, boolean calcChecksum) {
	packet = (int[]) p.clone();

	if (calcChecksum) {
		packet[packet.length - 3] = calcChecksum();
	}

	if (isLegal() != -1) {
		System.out.println("Error in byte: " + isLegal());
		throw (new InvalidPacketException(p, isLegal()));
	}
}

/**
* Calculates the checksum for the packet.
* Does <b> not </b> insert it into the correct position of the int[] packet array. <br/>
* The method assumes that the packet is a valid Garmin-packet with all values containing their final
* values.
*/
public int calcChecksum() {
	int sum = 0;
	for (int i = 1 ; i <= packet.length - 4 ; i++) {
		sum += packet[i];
	}

	sum = sum % 256;
	sum = sum ^ 255;
	sum += 1;
	return sum;
}

/**
* Returns the ID (ie. type) of the packet.
*/
public int getID() {
	return packet[1];
}

/**
* Returns the amount of bytes in the data-field of this packet.
*/
public int getDataLength() {
	return packet[2];
}

/**
* Returns the packet-byte at position i.
*/
protected int getByte(int i) {
	return packet[i];
}

/**
* Returns the packet in it's original byte-form.
* <br/><b> Note:</b> The array returned is a clone of the array contained in the class.
* Changing the values in the array will not affect the contents of the class.
*/
protected int[] getPacket() {
	return (int[]) packet.clone();
}

/**
* Returns the length of the entire packet in bytes.
*/
protected int getLength() {
	return packet.length;
}

/**
* Method that reads a Garmin-word in the packet and returns it as an int.
* This method can be used to read both int and word from a Garmin-packet.
*/
protected int readWord(int packet_index) {
	int sum = packet[packet_index++];
```

```
        sum += packet[packet_index++] << 8;
        return sum;
}

/**
 * Method that reads a Garmin-long in the packet and returns it as an int.
 */
protected int readLong(int packet_index) {
        int res = packet[packet_index++];
        res += packet[packet_index++] << 8;
        res += packet[packet_index++] << 16;
        res += packet[packet_index++] << 24;

        return res;
}

/**
 * Method that reads a null-terminated string.
 */
protected String readNullTerminatedString(int packet_index) {
        StringBuffer res = new StringBuffer(20);
        while ((packet[packet_index] != 0) && (packet_index != packet.length )) {
                res.append( (char) packet[packet_index++]);

        }
        return res.toString();
}

/**
 * Method that translates a packet-id into a human-readable string.
 */
public static String idToString(int id) {
        switch (id) {
                case Pid_Ack_Byte :
                        return "Acknowledge packet";
                case Pid_Command_Data :
                        return "Command packet";
                case Pid_Date_Time_Data :
                        return "Date and time data";
                case Pid_Nak_Byte :
                        return "Not acknowledged packet";
                case Pid_Product_Data :
                        return "Product data.";
                case Pid_Product_Rqst :
                        return "Product request";
                case Pid_Protocol_Array :
                        return "Protocol array packet";
                case Pid_Position_Data :
                        return "position data";
                case Pid_Pvt_Data :
                        return "PVT data";
                case Pid_Records :
                        return "Start of record transfer";
                case Pid_Wpt_Data :
                        return "waypoint data";
                default :
                        return "unknown data";
        }
}

/**
 * <i> Debug-method. </i>
 * Returns a String-representation of the bytes in the packet.
 */
public String getRawPacket() {
        StringBuffer s = new StringBuffer();
        for (int i = 0 ; i < packet.length ; i++)
                s.append(" " + packet[i]);
        return s.toString();
}

/**
 * This is a factory-method capable of creating instances the commandpackets from A010. (Device Command
Protocol 1)
 * returns null if it can't make a packet from the argument supplied.
 * <br/> <i>type<i/> can be one of the following constants:
 * <ul>
```

```
 * <li> Cmnd_Turn_Off_Pwr
 * <li> Cmnd_Transfer_Posn
 * <li> Cmnd_Transfer_Time
 * <li> Cmnd_Abort_Transfer
 * <li> Cmnd_Transfer_Alm
 * <li> Cmnd_Transfer_Prx
 * <li> Cmnd_Transfer_Rte
 * <li> Cmnd_Transfer_Trk
 * <li> Cmnd_Transfer_Wpt
 * <li> Cmnd_Start_Pvt_Data
 * <li> Cmnd_Stop_Pvt_Data
 * </ul>
 */
 public static GarminPacket createCommandPacket(int type) {
        switch (type) {
                case Cmnd_Turn_Off_Pwr :
                case Cmnd_Transfer_Posn:
                case Cmnd_Transfer_Time:
                case Cmnd_Abort_Transfer :
                case Cmnd_Transfer_Alm :
                case Cmnd_Transfer_Prx :
                case Cmnd_Transfer_Rte :
                case Cmnd_Transfer_Trk :
                case Cmnd_Transfer_Wpt :
                case Cmnd_Start_Pvt_Data :
                case Cmnd_Stop_Pvt_Data       :
                        return new GarminPacket(new int[] {DLE, Pid_Command_Data, 2, type,0 , 0, DLE,
ETX}, true);
                default :
                        return null;
        }
 }

 /**
 * This method is capable of making the data-packets from L000 (basic link protocol).
 * <br/> <i>type</i> can be one of the following constants:
 * <ul>
 * <li> Pid_Ack_Byte
 * <li> Pid_Nak_Byte
 * <li> Pid_Protocol_Array
 * <li> Pid_Product_Rqst
 * <li> Pid_Product_Data
 * </ul>
 * The argument <i> data </i> is an array of int that will be put in the data-field of the packet.
 */
 public static GarminPacket createBasicPacket(int type, int[] data) {
        switch (type) {
                case Pid_Ack_Byte :
                case Pid_Nak_Byte :
                case Pid_Protocol_Array :
                case Pid_Product_Rqst:
                case Pid_Product_Data :
                        int[] packet = new int[data.length + 6];
                        packet[0] = DLE; packet[1] = type;
                        packet[2] = data.length;
                        System.arraycopy(data, 0, packet, 3, data.length);
                        packet[packet.length - 3] = 0;
                        packet[packet.length - 2] = DLE;
                        packet[packet.length - 1] = ETX;
                        return new GarminPacket(packet, true);
                default :
                        return null;
        }
 }

 /**
 * Checks if the packet is valid with regards to header, footer,data-field-length and checksum.
 * Returns the index of the illegal byte. If packet is ok, -1 is returned.
 */
 public int isLegal() {
        if (packet[0] != DLE)
                return 0;

        int size = packet[2];

        if (size + 6 != packet.length)
                return 2;
```

```
        if ( packet[packet.length  - 3] != calcChecksum() )
                return packet.length  - 3;

        if ( packet[packet.length  - 2] != DLE )
                return packet.length  - 2;

        if ( packet[packet.length  - 1] != ETX )
                return packet.length  - 1;

        return -1;
  }

  /**
  * Method that reads a Garmin-byte in the packet and returns it as a short.
  */
  protected short readByte(int packet_index) {
        return (short) packet[packet_index];
  }

  /**
  * Method that reads a Garmin-double in the packet and returns it as a double.
  */
  protected double readDouble(int packet_index) {

        long res = 0;

        res += ( (long) packet[packet_index++] );
        res += ( (long) packet[packet_index++] ) << 8;
        res += ( (long) packet[packet_index++] ) << 16;
        res += ( (long) packet[packet_index++] ) << 24;
        res += ( (long) packet[packet_index++] ) << 32;
        res += ( (long) packet[packet_index++] ) << 40;
        res += ( (long) packet[packet_index++] ) << 48;
        res += ( (long) packet[packet_index++] ) << 56;

        return Double.longBitsToDouble(res);
  }

  /**
  * Returns a human-readable string with information to the packet's contents.
  */

  public String toString() {
        return "GarminPacket containing " + idToString(getID());
  }

  /**
  * Method that reads a Garmin-float in the packet and returns it as a float.
  */
  protected float readFloat(int packet_index) {
        int res = 0;
        res += packet[packet_index++];
        res += packet[packet_index++] << 8;
        res += packet[packet_index++] << 16;
        res += packet[packet_index++] << 24;

        return Float.intBitsToFloat(res);
  }


}
```

10.2.3PositionDataPacket

```
package dk.itu.haas.GPS.Garmin;

import dk.itu.haas.GPS.*;

public class PositionDataPacket extends GarminPacket implements IPosition{
  private PositionRadians lat, lon;

  /**
```

```
  * Treats the packet p as a packet containing position-data.
  * Throws PacketNotRecognizedException if p is not a position-data-packet.
  * Throws InvalidPacketException if the packet contains too little data.
  */
  public PositionDataPacket(int[] p) {
          super(p);
          if (getID() != Pid_Position_Data) {
                  throw(new PacketNotRecognizedException(Pid_Position_Data, getID()));
          }

          if (getDataLength() != 16) {
                  throw(new InvalidPacketException(packet, 2));
          }

          lat = new PositionRadians(readDouble(3));
          lon = new PositionRadians(readDouble(11));
  }

  /**
  * This method is a copy-constructor allowing to "upgrade" a GarminPacket to a PositionPacket.
  * Throws PacketNotRecognizedException if p is not a position-data-packet.
  */
  public PositionDataPacket(GarminPacket p) {
          this( p.packet );
  }

  /**
  * This method returns the latitude of the position.
  */
  public PositionRadians getLatitude() {
          return lat;
  }

  /**
  * This method returns the longitude of the position.
  */
  public PositionRadians getLongitude() {
          return lon;
  }

  /**
  * Returns a String containing the position in a human-readable format.
  */
  public String toString() {
          StringBuffer res = new StringBuffer();
          res.append("\nPosition:");
          res.append("\nLatitude: " + lat.toString());
          res.append("\nLongitude: " + lon.toString());
          return res.toString();
  }
}
```

10.2.4 PVTDataPacket

```
package dk.itu.haas.GPS.Garmin;
import dk.itu.haas.GPS.*;
/**
* This class encapsulates the PVT (Position, velocity and time) packet.
* After receiving a Cmnd_Start_Pvt-packet, the GPS will continually transmit
* packets of the PVT-type.
*/

public class PVTDataPacket extends GarminPacket implements IPosition, ITime {

  protected float alt;                  // Altitude
  protected float epe;                  // Estimated position error
  protected float eph;                  // Horizontal epe
  protected float epv;                  // Vertical epe
  protected int fix;                            // Position fix
  protected double  tow;                // Time of week (seconds).
  protected PositionRadians lat;        // Latitude
  protected PositionRadians lon;        // Longitude
  protected float veast;                        // Velocity east.
  protected float vnorth;                       // Velocity north.
  protected float vup;                  // Velocity up.
```

```
protected float msl_hght;            //
protected int leap_scnds;            // Time difference between GPS and GMT (UTC)
protected long wn_days;                      // Week number days.

/**
* Treats the packet p as a packet containing PVT-data.
* Throws PacketNotRecognizedException if p is not a PVT-packet.
* Throws InvalidPacketException if the packet contains too little data.
*/
public PVTDataPacket(int[] p) {
        super(p);

        if (getID() != Pid_Pvt_Data) {
                throw(new PacketNotRecognizedException(Pid_Pvt_Data, getID()));
        }

        if (getDataLength() != 64) {
                throw(new InvalidPacketException(packet, 2));
        }

        alt = readFloat(3);
        epe = readFloat(7);
        eph = readFloat(11);
        epv = readFloat(15);
        fix = readWord(19);
        tow = readDouble(21);
        lat = new PositionRadians( readDouble(29));
        lon = new PositionRadians( readDouble(37));
        veast = readFloat(45);
        vnorth = readFloat(49);
        vup = readFloat(53);
        msl_hght = readFloat(57);
        leap_scnds = readWord(61);
        wn_days = readLong(63);
}

/**
* This method is a copy-constructor allowing to "upgrade" a GarminPacket to a PVTDataPacket.
* Throws PacketNotRecognizedException if p is not a PVT-data-packet.
*/
public PVTDataPacket(GarminPacket p) {
        this( p.packet );
}


/** Returns the hour of the day. */
public int getHours() {
        int hour = (int) tow;
        hour = hour % (24 * 60 * 60); // Remove all preceding days.
        hour = hour / 3600;
        return hour;
}

/** Returns the minute of the hour. */
public short getMinutes() {
        int minute = (int) tow;
        minute = minute % (24 * 60 * 60); // Remove all preceding days.
        minute = minute / 60;
        minute = minute % 60;
        return (short) minute;
}

/** Returns the second of the minute. */
public short getSeconds() {
        return (short) (tow % 60);
}

/**
* This method returns the latitude of the position.
*/
public PositionRadians getLatitude() {
        return lat;
}

/**
* This method returns the longitude of the position.
*/
```

```
  public PositionRadians getLongitude() {
          return lon;
  }
}
```

10.2.5TimeDataPacket

```
package dk.itu.haas.GPS.Garmin;

import dk.itu.haas.GPS.*;

/**
* This class encapsulates the information of a Garmin-Date-Time-packet.
*/

public class TimeDataPacket extends GarminPacket implements ITime, IDate{
  /** Month (1-12) */
  protected short month;
  /** Day (1-31) */
  protected short day;
  /** Year. */
  protected int year;
  /** Hour of the day. */
  protected int hour;
  /** Minute of the hour. */
  protected short minute;
  /** Second of the minute. */
  protected short second;

  /**
  * Treats the packet p as a packet containing Time-data.
  * Throws PacketNotRecognizedException if p is not a Time-packet.
  * Throws InvalidPacketException if the packet contains too little data.
  */
  public TimeDataPacket (int[] p) {
          super(p);

          if (getID() != Pid_Date_Time_Data) {
                  throw(new PacketNotRecognizedException(Pid_Date_Time_Data, getID()));
          }

          if (getDataLength() != 8) {
                  throw(new InvalidPacketException(packet, 2));
          }

          month = readByte(3);
          day = readByte(4);
          year = readWord(5);
          hour = readWord(7);
          minute = readByte(9);
          second = readByte(10);
  }

  /**
  * Treats the packet p as a packet containing Time-data.
  * Throws PacketNotRecognizedException if p is not a Time-packet.
  * Throws InvalidPacketException if the packet contains too little data.
  */
  public TimeDataPacket (GarminPacket p) {
          this(p.packet);
  }

  /** Returns the day of the month. */
  public short getDay() {
          return day;
  }

  /** Returns the month. */
  public short getMonth() {
          return month;
  }

  /** returns the year. */
  public int getYear() {
```

```
        return year;
  }

  /** Returns the hour of the day. */
  public int getHours() {
        return hour;
  }

  /** Returns the minute of the hour. */
  public short getMinutes() {
        return minute;
  }

  /** Returns the second of the minute.*/
  public short getSeconds() {
        return second;
  }

  /**
  * Returns the value of this packet in a human-readable format.
  */
  public String toString() {
        StringBuffer res = new StringBuffer();
        if (hour < 10)
              res.append("0");
        res.append(hour +":");
        if (minute < 10)
              res.append("0");
        res.append(minute +":");
        if (second < 10)
              res.append("0");
        res.append(second + " on ");
        res.append(day + "/");
        res.append(month + "-" + year);
        return res.toString();
  }
}
```

10.2.6WaypointDataPacket

```
package dk.itu.haas.GPS.Garmin;

import dk.itu.haas.GPS.*;

/**
* This class encapsulates a Waypoint-packet. The Garmin-protocol contains a huge amount of different
* Waypoint-Packet specifications. Only the one labelled D108 is implemented so far.
*/
public class WaypointDataPacket extends GarminPacket implements IWaypoint{
  /**
  * Holds information about which waypoint-format this Garmin-unit uses. The default is 108.
  */
  protected static int datatypeversion = 108;

  /** Class of waypoint. */
  protected short wpt_class;
  /** Color of waypoint when displayed on the GPS.*/
  protected short color;
  /** Display options.*/
  protected short dspl;
  /** Attributes. */
  protected short attr;
  /** Waypoint symbol. */
  protected int smbl;
  /** Subclass of waypoint */
  protected short[] subclass;
  /** Latitude of waypoint. */
  protected PositionDegrees lat;
  /** Longitude of waypoint. */
  protected PositionDegrees lon;
  /** Altitude. */
  protected float alt;
  /** Depth. */
  protected float depth;
  /** Proximity distance in meters. */
```

```
protected float dist;
/** State.*/
protected char[] state;
/** Country code.*/
protected char[] cc;
/** Waypoint name. */
protected String name;
/** Waypoint comment. */
protected String comment;
/** facility name. */
protected String facility;
/** City name. */
protected String city;
/** Address number */
protected String address;
/** Intersecting road label.*/
protected String cross_road;

/**
* Throws a PacketNotRecognizedException if the Waypoint-dataformat is not implemented.
*/

public WaypointDataPacket(int[] p) {
        super(p);

        if (getID() != Pid_Wpt_Data) {
                throw(new PacketNotRecognizedException(Pid_Wpt_Data, getID()));
        }

        switch (datatypeversion) {
                case 108 :
                        initD108();
                        break;
                default :
                        System.out.println("Waypoint-type " + datatypeversion + " not supported.");
                        throw(new PacketNotRecognizedException(Pid_Wpt_Data, getID()));
        }
}

public WaypointDataPacket(GarminPacket p) {
        this( (int[]) p.packet.clone() );
}

/**
* Configures this packet as a D108 (Waypoint).
*/
private void initD108() {
        long l;
        l  = readLong(27);
        // Calculate from semicircles to degrees.
        lat = new PositionDegrees( (l * 180) / Math.pow(2.0d, 31.0d) );
        l = readLong(31);
        lon = new PositionDegrees( (l * 180) / Math.pow(2.0d, 31.0d) );
        name = readNullTerminatedString(49);
}


/**
* Sets which version of the packet that this class should treat.
* <br/><b> Note: </b>Setting this value will affect all instances of the class.
*/
public static void setDatatypeVersion(int v) {
        datatypeversion = v;
}

/**
* This method returns the latitude of the waypoint.
*/
public PositionRadians getLatitude() {
        return lat.convertToRadians();
}

/**
* This method returns the longitude of the waypoint.
*/
public PositionRadians getLongitude() {
        return lon.convertToRadians();
```

```
  }

  /**
  * This method returns the name of the waypoint.
  */
  public String getName() {
          return name;
  }
}
```

10.2.7ProductDataPacket

```
package dk.itu.haas.GPS.Garmin;

public class ProductDataPacket extends GarminPacket {
  /** Product-ID of GPS. */
  protected int productID;
  /** Software version in GPS.*/
  protected int SWversion;
  /** Description of GPS as given by GPS. */
  protected String productDesc;

  /**
  * Treats the packet p as a packet containing product-data.
  * Throws PacketNotRecognizedException if p is not a product-data-packet.
  */
  public ProductDataPacket(int[] p) {
          super(p);
          if (getID() != Pid_Product_Data) {
                  throw(new PacketNotRecognizedException(Pid_Product_Data, getID()));
          }

          productID = readWord(3);
          SWversion = readWord(5);
          productDesc = readNullTerminatedString(7);
  }

  /**
  * This method is a copy-constructor allowing to "upgrade" a GarminPacket to a ProductDataPacket.
  * Throws PacketNotRecognizedException if p is not a product-data-packet.
  */
  public ProductDataPacket(GarminPacket p) {
          this( p.packet );
  }


  /** Returns the product ID of the GPS.*/
  public int getProductID() {
          return productID;
  }

  /** Returns the version of the software in the GPS. */
  public int getSWVersion() {
          return SWversion;
  }

  /** Returns the supplied description of the GPS. */
  public String getDescription() {
          return productDesc;
  }
  /** Returns a human-readable version of this packet. */
  public String toString() {
          StringBuffer res = new StringBuffer();
          res.append(productDesc + '\n');
          res.append("Product ID: " + productID);
          res.append("\nSoftware version: " + SWversion);
          return res.toString();
  }

}
package dk.itu.haas.GPS.Garmin;
```

10.2.8ProtocolDataPacket

```
public class ProtocolDataPacket extends GarminPacket {

  protected char[] tags;
  protected int[]  data;

  /**
  * Treats the packet p as a packet containing data about which protocols the GPS support.
  * Throws PacketNotRecognizedException if p is not a product-data-packet.
  */
  public ProtocolDataPacket(int[] p) {
          super(p);
          if (getID() != Pid_Protocol_Array) {
                  throw(new PacketNotRecognizedException(Pid_Protocol_Array, getID()));
          }

          if (getDataLength() % 3 != 0) {
                  throw(new InvalidPacketException(packet, 2));
          }

          tags = new char[getDataLength() / 3];
          data = new int[getDataLength() / 3];

          int packet_index = 3;
          int array_index = 0;
          while (packet_index != getDataLength() + 3) {
                  tags[array_index] = (char) readByte(packet_index++);
                  data[array_index] = readWord(packet_index);
                  packet_index += 2;
                  array_index++;
          }
  }

  public ProtocolDataPacket(GarminPacket p) {
          this(p.packet);
  }

  /**
  * This method will return the exact version of a protocol.
  * If the protocol is not supported by the GPS, the method returns -1.
  */
  public int getVersion(char tag, int protocol) {
          for (int i = 0 ; i < tags.length ; i++) {
                  if (tags[i] == tag) {
                          if ( data[i] / protocol == 1)
                                  return data[i];
                  }
          }
          return -1;
  }

  public String toString() {
          StringBuffer res = new StringBuffer();
          res.append("Tag:\tData:\n");
          for (int i = 0 ; i < tags.length ; i++) {
                  res.append(tags[i] + "\t" + data[i] + '\n');
          }
          return res.toString();
  }
}
```

10.2.9RecordsPacket

```
package dk.itu.haas.GPS.Garmin;

/**
* This packet is transmitted between devices before a large transfer of data-units, ie. a transfer of
waypoints.
*
*/
public class RecordsPacket extends GarminPacket {
  /** The number of records to come, that this packet announces. */
  protected int number;
```

```
  public RecordsPacket(int[] p) {
          super(p);

          if (getID() != Pid_Records) {
                  throw(new PacketNotRecognizedException(Pid_Records, getID()));
          }

          if (getDataLength() != 2) {
                  throw(new InvalidPacketException(packet, 2));
          }

          number = readWord(3);
  }

  public RecordsPacket(GarminPacket p) {
          this(p.packet);
  }

  /** Returns the number of records that this packet announces. */
  public int getNumber() {
          return number;
  }
}
```

10.2.10GarminInputStream

```
package dk.itu.haas.GPS.Garmin;
import java.io.*;

/**
* This class provides the functionality of automatically removing the double DLEs from the GPS-
inputstream.
* The double-DLEs can be found in the size-,data-, and checksum-fields.
* The only method providing the filtering-functionality is read().
*/
public class GarminInputStream extends FilterInputStream {
  /*
  * Last value read.
  */
  private int prev;

  /**
  * Takes the stream to the GPS-unit as an argument.
  */
  public GarminInputStream(InputStream i) {
          super(i);
          in = i;
          prev = 0;
  }

  /**
  * Reads a packet from the stream. <br/>
  * <b> Note: </b> Method assumes that it's called between packets, ie. when the first byte of a packet
is the
  * next in the stream. If this condition is met, the method will leave the stream in the same state.
  */
  public int[] readPacket() throws InvalidPacketException, IOException {
          int c;
          int[] packet;
          int id, size;
          c = read();

          if (c != GarminPacket.DLE) {
                  throw (new InvalidPacketException( new int[] { GarminPacket.DLE }, 0));
          }

          id = read();
          size = read();
          packet = new int[size + 6];
          packet[0] = GarminPacket.DLE;
          packet[1] = id;
          packet[2] = size;
          for (int i = 0 ; i < size + 3 ; i++)
```

```
                packet[3 + i] = read();

        if (packet[packet.length - 2] != GarminPacket.DLE) {
                throw (new InvalidPacketException(packet, packet.length - 2));
        }

        if (packet[packet.length - 1] != GarminPacket.ETX) {
                throw (new InvalidPacketException(packet, packet.length - 1));
        }
        return packet;
  }

  /**
  * Returns the next byte from the stream. Makes sure to remove DLE stuffing.
  */
  public int read() throws IOException{
        int c = in.read();
        if ( prev == 16 && c == 16)
                return prev = in.read();
        else
                return prev = c;
  }
}
```

10.2.11GarminOutputStream

```
package dk.itu.haas.GPS.Garmin;
import java.io.*;

/**
* This class take care of adding DLE-stuffing to all packets sent to the GPS.
* <b> NOTE: </b> Only the method write(GarminPacket) performs addition of DLE-stuffing. The remaining
* methods write directly to the GPS without format-control.
*/

public class GarminOutputStream extends FilterOutputStream {
  public GarminOutputStream(OutputStream o) {
        super(o);
  }

  public synchronized void write (GarminPacket packet) throws IOException, InvalidPacketException {
        if (packet.isLegal() != -1) {
                throw (new InvalidPacketException(packet.getPacket(), packet.isLegal()));
        }
        write(packet.getByte(0));
        write(packet.getByte(1));

        int c;
        // Iterate through size-field, data-field and checksum-field. Add stuffing where necessary.
        for (int i = 0 ; i < packet.getByte(2) + 2 ; i++) {
                c = packet.getByte(i + 2);
                write(c);
                if (c == GarminPacket.DLE)
                        write(c);
        }

        write(GarminPacket.DLE);
        write(GarminPacket.ETX);
        flush();
  }
}
```

10.2.12GarminListener

```
package dk.itu.haas.GPS.Garmin;

/**
 This interface should be implemented by classes that are interested in getting all the Garmin-packets
 transmitted by a Garmin-GPS. Listener's will receive all packets including ACKs and NAKs. Only
exception
 are malformed packets.
*/

public interface GarminListener {
```

```
  /** This method will be called for each packet received from the GPS. */
  public void GarminPacketReceived(GarminPacket p);
}
```

10.2.13 InvalidPacketException

```
package dk.itu.haas.GPS.Garmin;

/**
* This method is thrown from the constructors of the packet-classes, whenever the int[]-array is not 
formatted according to the Garmin-packet-specs.
*/
public class InvalidPacketException extends RuntimeException {

  private int[] packet;
  private int index;

  /**
  * Creates an InvalidPacketException. pack is a reference to the byte-array that caused the exception.
  * i is the index of the byte that was in error.
  */
  public InvalidPacketException(int[] pack, int i) {
        packet = pack;
        index = i;
  }

  /**
  * Returns the packet that caused the exception to be thrown.
  * <b>Note:</b> The return-value is a reference to the original array. Changes will likely propagate
  * to other parts of the program.
  */
  public int[] getPacket() {
        return packet;
  }

  /**
  *
  */

  /**
  * Returns the index of the first erroneously configured byte.
  */
  public int getIndex() {
        return index;
  }

  /**
  * Returns a formatted string showing which byte is wrong.
  */
  public String toString() {
        StringBuffer res = new StringBuffer();
        res.append("\nByte\tvalue\n");
        for (int i = 0 ; i < packet.length ; i++) {
              res.append(" " + i+"\t"+packet[i]);
              if (i == index)
                    res.append(" <- Erroneous");
              res.append('\n');
        }
        return res.toString();
  }
}
```

10.2.14 PacketNotRecognizedException

```
package dk.itu.haas.GPS.Garmin;

/**
* This exception is thrown whenever a method expects one type of packet, but receives another.
* An example is if a PositionDataPacket is initialized with the byte-array containing time-data.
* This exception is a runtime exception because it's assumed that there will in most cases be type-
checking
* of packets.
*/
```

```
public class PacketNotRecognizedException extends RuntimeException {

  /**
  * expected is the type of packet that the method was expecting. actual is the type of the packet that
  * it received.
  */
  public PacketNotRecognizedException(int expected, int actual) {
          super(  "\nPacket initialization error:\n" + "Expected: " +
                          GarminPacket.idToString(expected) + '\n' +
                          "received: " + GarminPacket.idToString(actual) + '\n');
  }
}
```

10.2.15UnknownPacketException

```
package dk.itu.haas.GPS.Garmin;

public class UnknownPacketException extends Exception {

}
```

10.3dk.itu.haas.GPS.services –package

10.3.1AreaAlarm

```
package dk.itu.haas.GPS.services;
import dk.itu.haas.GPS.*;
import java.util.Vector;

/**
* This class implements an AreaAlarm-service. The class allows the user to specify two positions, which
* will be used as opposite corners in a rectangular area. Whenever the GPS enters or exits the area all
* listeners are notified through the IAlarmListener-interface.
*/
public class AreaAlarm implements IGPSlistener {
  private GPS gps;
  // The positions defining the rectangle to be used.
  private PositionRadians bottom_longitude,
                                              top_longitude,
                                              left_latitude,
                                              right_latitude;

  private Vector alarmListeners;
  private boolean inside;

  public AreaAlarm(GPS g, Position p1, Position p2) {
          gps = g;
          gps.setAutoTransmit(true);
          gps.addGPSlistener(this);

          alarmListeners = new Vector();

          PositionRadians l1, l2;
          l1 = p1.getLatitude();
          l2 = p2.getLatitude();
          if (l1.equals(l2))
                  throw (new RuntimeException("No area. Latitude of point 1 equals latitude of point
2."));

          if (l1.greaterThan(l2)) {
                  right_latitude = l1;
                  left_latitude = l2;
          } else {
                  right_latitude = l2;
                  left_latitude = l1;
          }

          l1 = p1.getLongitude();
          l2 = p2.getLongitude();
          if (l1.equals(l2))
                  throw (new RuntimeException("No area. Longitude of point 1 equals longitude of point
2."));

          if (l1.greaterThan(l2)) {
```

```
			top_longitude = l1;
			bottom_longitude = l2;
		} else {
			top_longitude = l2;
			bottom_longitude = l1;
		}
	}

	/**
	* Adds l to the list of listeners interested in receiving notification when the GPS enters or exits
the area.
	*/
	public void addAlarmListener(IAlarmListener l) {
		// Only allow a listener to be registered once.
		if (alarmListeners.contains(l))
			return;

		alarmListeners.add(l);
		return;
	}

	/**
	* Removes the the Alarm-listener l from the list of Waypoint-listeners.
	*/
	public void removeAlarmListener(IAlarmListener l) {
		while (alarmListeners.removeElement(l)) {}
	}


	/**
	* This method propagates the information that the gps has exited the area to all listeners.
	*/
	protected void fireOutside() {
		for (int i = 0 ; i < alarmListeners.size() ; i++) {
			((IAlarmListener) alarmListeners.elementAt(i)).exitedAlarm();
		}
	}

	/**
	* This method propagates the information that the gps has entered the area to all listeners.
	*/
	protected void fireInside() {
		for (int i = 0 ; i < alarmListeners.size() ; i++) {
			((IAlarmListener) alarmListeners.elementAt(i)).enteredAlarm();
		}
	}

	public void timeReceived(ITime t) {}

	public void dateReceived(IDate d) {}

	public void positionReceived(IPosition pos) {
		System.out.println("Areaalarm: Received pos. Analyzing!");
		if (inside == false) {
			if (	(pos.getLatitude().greaterThan(left_latitude)) &&
					(pos.getLatitude().smallerThan(right_latitude)) &&
					(pos.getLongitude().greaterThan(bottom_longitude)) &&
					(pos.getLongitude().smallerThan(top_longitude))) {
				inside = true;
				fireInside();
			}
		} else {
			if (	!((pos.getLatitude().greaterThan(left_latitude)) &&
					(pos.getLatitude().smallerThan(right_latitude)) &&
					(pos.getLongitude().greaterThan(bottom_longitude)) &&
					(pos.getLongitude().smallerThan(top_longitude)))) {
				inside = false;
				fireOutside();
			}
		}
	}
}
```

10.3.2IAlarmListener

```
package dk.itu.haas.GPS.services;

/**
* This interface allows a class to listen on an AreaAlarm.
*/
public interface IAlarmListener {
  /**
  * This method is called by the AreaAlarm when the GPS exists the area.
  */
  public void exitedAlarm();

  /**
  * This method is called by the AreaAlarm when the GPS enters the area.
  */
  public void enteredAlarm();
}
```

10.4dk.itu.haas.GPS.examples –package

10.4.1ConnectionTest

```
package dk.itu.haas.GPS.examples;
import dk.itu.haas.GPS.*;
import dk.itu.haas.GPS.Garmin.*;

import javax.comm.*;
import java.util.Enumeration;
import java.util.Vector;
import java.io.*;

/**
* A simple test-program that queries the GPS for a description of itself. <br/>
* A good way to initially test the connection to the GPS. <br/>
* Attempts to turn off the GPS after retrieving the description.
*/

class ConnectionTest {
  /** The communication port being used. */
  String portname;
  BufferedReader inuser;
  GPS gps;
  CommPort port;
  BufferedInputStream input;
  BufferedOutputStream output;

  public static void main(String args[]) {
          new ConnectionTest();
  }

  public ConnectionTest() {
          inuser = new BufferedReader(new InputStreamReader(System.in));
          String portname = ListPorts();
          String gpsbrand = ListGPS();

          try {
                  port = CommPortIdentifier.getPortIdentifier(port).open("ConnectionTest", 3000);
          } catch (NoSuchPortException e) {
                  System.out.println("Port not found!");
                  return;
          } catch (PortInUseException e) {
                  System.out.println("Port already in use by " + e.currentOwner);
                  return;
          }

          try {
                  input = new BufferedInputStream(port.getInputStream());
                  output = new BufferedOutputStream(port.getOutputStream());
          } catch (IOException e) {
                  System.out.println("Error opening port " + portname);
                  return;
```

```
        }
        if (gpsbrand.equals("GARMIN") ) {
                gps = new GarminGPS(input, output);
        }
        try {
                Thread.sleep(1500);
        } catch (InterruptedException e) {}

        System.out.println("Connected to GPS:");
        System.out.println( gps.getDescription());
        gps.shutdown(false);
}

/**
* Ok. Ridiculous method, but I'm hoping that more GPS-types will be added later.
* Lists the types of GPS that are available and lets the user pick one.
*/
private String ListGPS() {
        int index = -1;
        while (index == -1) {
                System.out.println("\nAvailable GPS-types:");
                System.out.println(" 1. Garmin");
                System.out.print("Select GPS: ");
                String input = readFromUser();

                try {
                        index = Integer.parseInt(input);
                } catch (NumberFormatException e) {
                        index = -1;
                        continue;
                }

                if (index != 1) {
                        index = -1;
                        continue;
                }
        }
        switch (index) {
                case 1 :
                        return "GARMIN";
                default :
                        return "unknown";
        }
}

private String ListPorts() {
        Vector names = null;
        int index = -1;
        while (index == -1) {
                int j = 1;
                names = new Vector();
                System.out.println("Available ports: ");
                CommPortIdentifier c;
                for (Enumeration i = CommPortIdentifier.getPortIdentifiers() ; i.hasMoreElements() ;)
{
                        c = (CommPortIdentifier) i.nextElement();
                        System.out.print(j++ +". " + c.getName());
                        names.add(c.getName());
                        if (c.getPortType() == CommPortIdentifier.PORT_SERIAL)
                                System.out.print("\t SERIAL\n");
                        if (c.getPortType() == CommPortIdentifier.PORT_PARALLEL)
                                System.out.print("\t PARALLEL\n");
                }
                System.out.print("Select port: ");
                String input = readFromUser();

                try {
                        index = Integer.parseInt(input);
                } catch (NumberFormatException e) {
                                index = -1;
                                continue;
                }

                if ( (index < 1) || (index  >= names.size()) ) {
                        index = -1;
                        continue;
                }
```

```
        }

        return names.elementAt(index - 1).toString();
  }

  public String readFromUser() {
        try {
              return inuser.readLine();
        } catch (IOException e) {
              return "";
        }
  }
}
```

10.4.2AreaAlarmDemo

```
package dk.itu.haas.GPS.examples;
import dk.itu.haas.GPS.*;
import dk.itu.haas.GPS.services.*;
import dk.itu.haas.GPS.Garmin.*;
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import java.io.*;
import javax.comm.*;

public class AreaAlarmDemo extends JFrame implements ActionListener, IGPSlistener, IAlarmListener {
  GPS gps;
  AreaAlarm alarm;
  Position pos1 = null, pos2 = null;
  IPosition current = null;
  JLabel indicator;
  JButton mark1;
  JButton mark2;
  BufferedInputStream input;
  BufferedOutputStream output;


  public static void main(String args[]) {
        new AreaAlarmDemo();
  }

  public void actionPerformed(ActionEvent e) {
        if ( (e.getActionCommand().equals("mark1")) && (current != null)) {
              mark1.setEnabled(false);
              pos1 = new Position(current);
              current = null;
        }

        if ( (e.getActionCommand().equals("mark2")) && (current != null)) {
              mark2.setEnabled(false);
              pos2 = new Position(current);
              current = null;
        }

        if ( (pos1 != null) && (pos2 != null) ) {
              indicator.setText("Not inside area.");
              alarm = new AreaAlarm(gps, pos1, pos2);
              alarm.addAlarmListener(this);
              gps.removeGPSListener(this);
        }
  }


  public void exitedAlarm() {
        indicator.setText("Not inside area.");
  }

  public void enteredAlarm() {
        indicator.setText("Inside area.");
  }

  public AreaAlarmDemo() {
        super("AreaAlarm-demonstration");
```

```
        CommPort port;

        try {
               port = CommPortIdentifier.getPortIdentifier("COM1").open("AreaAlarmDemo", 3000);
        } catch (NoSuchPortException e) {
               System.out.println("COM1 not found!");
               return;
        } catch (PortInUseException e) {
               System.out.println("Port already in use by " + e.currentOwner);
               return;
        }

        try {
               input = new BufferedInputStream(port.getInputStream());
               output = new BufferedOutputStream(port.getOutputStream());
        } catch (IOException e) {
               System.out.println("Error opening COM1");
               return;
        }

        gps = new GarminGPS(input, output);
        gps.setAutoTransmit(true);
        gps.addGPSlistener(this);


        indicator = new JLabel("Nothing recorded yet.", JLabel.CENTER);
        getContentPane().setLayout(new BorderLayout());

        mark1 = new JButton("Mark position 1");
        mark1.setActionCommand("mark1");
        mark1.addActionListener(this);
        mark2 = new JButton("Mark position 2");
        mark2.setActionCommand("mark2");
        mark2.addActionListener(this);


        getContentPane().add(mark1, BorderLayout.NORTH);
        getContentPane().add(mark2, BorderLayout.SOUTH);
        getContentPane().add(indicator, BorderLayout.CENTER);

        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(320,200);
        setLocationRelativeTo(null);
        show();

        System.out.println(gps.getDescription());
  }

  public void timeReceived(ITime t) {}
  public void dateReceived(IDate d) {}

  public void positionReceived(IPosition pos) {
        System.out.println("Received!");
        current = pos;
  }
}
```