Semester Thesis

# GPS on iPAQ

Philip Frey
freyp@student.ethz.ch

Dept. of Computer Science
Swiss Federal Institute of Technology (ETH) Zurich

Prof. Dr. Roger Wattenhofer
Distributed Computing Group
Supervisor: Aaron Zollinger

Winter 2004/05

# Contents

# Chapter 1

# Introduction

One of the most commonly used positioning systems today is the Global Positioning System (GPS). The goal of this semester thesis is to acquire information from this system and save as well as display them in an appropriate manner on an iPAQ device. In particular I used a Compaq iPAQ 3870 PDA (portable digital assistant) in combination with a HAiCOM GPS receiver [6].
I started parsing the data from the GPS receiver. After that my task was to develop some applications that operate on the GPS API written in the first step and in the end to determine how accurate and reliable this system is. The applications should display as well as trace the position information in an appropriate way.
All of the software is written in Java since it is then possible to run the same program not only on the iPAQ but on any system with a serial port (e.g. PCMCIA or CompactFlash interface for the HAiCOM GPS receiver to fit in) and a Java Virtual Machine (JVM) version 1.1 or higher. The two disadvantages of this approach were the slightly limited JVM available for PDAs and the loss in speed that comes with its additional abstraction layer.

# Chapter 2

# GPS

The system was originally developed by the U.S. Department of Defense [2] for military purposes only. Until May 2000 timing errors were inserted to limit the accuracy to 100-200m. Without this so-called selective availability it is now possible to determine your position within a range of less than 10m if enough satellites are available. There are at least 24 operating satellites that orbit the earth in 12 hours. Their constellation is such that from any point on earth you should be able to get signals from usually five to eight satellites at a time. A line of sight between the receiver and the satellite is required in order to use it.


Figure 2.1: GPS Constellation

For more information about GPS see [3] and [4].

## 2.1 HAiCOM GPS Receiver

For this thesis I used a GPS receiver by HAiCOM [6]. It is mainly a CompactFlash extension card that comes with an adapter for the PCMCIA interface and therefore is treated as a serial interface in Java. The iPAQ had to be equipped with a PCMCIA jacket for this purpose. The receiver comes without much software. All I had was a small application to test whether the device was working and to determine the correct COM port for it. The GPS information is given in the NMEA v2.2 format (also known as NMEA0183). The receiver produces an updated version of each part of the NMEA0183 protocol every second. For detailed specifications see [7].

### 2.1.1 NMEA 0183

NMEA stands for *National Marine Electronics Association* [5] and consists of a standardized set of giving info 'sentences' from which I used the following:

**GGA** (Global Positioning System Fix Data)

**GSA** (GPS Dilution of Precision and Active Satellites)

**GSV** (GPS Satellites in View)

**RMC** (Recommended Minimum Specific GPS Data)

The most important fields of the **GGA** sentence are the *UTC* (Coordinated Universal Time), *latitude*, *longitude*, *altitude* and the current *dilution of precision*.
The **GSA** sentence gives information about the satellites that are currently used for the determination of the position. All visible satellites then are listed in the **GSV** sentence and the **RMC** gives a kind of summary including *UTC* and *date*, *latitude*, *longitude*, *speed* and *course*. This is the most interesting information for "everyday life".

## 2.2 Accuracy and Reliability of the GPS Receiver

### 2.2.1 Accuracy: Position, Height and Speed

According to the manufacturer of the GPS receiver (HAiCOM) we should be able to determine where we are within a radius of 10m in 95% of the cases. For the velocity they specify an accuracy of 0.1 meters per second (0.36km/h). The time should be within 1 microsecond synchronized to the GPS time. In addition to this I did my own tests. I drove and walked around taking samples in constant intervals. The GPS receiver indicates the accuracy in terms of Horizontal Dilution of Precision (HDOP), Position Dilution of Precision (PDOP), Vertical Dilution of Precision (VDOP). DOP is a mathematical representation for the quality of the GPS position solution. The lower these values are the better is the accuracy (the optimum is 1). The following three sections present results.

#### Position

The accuracy highly depends on how many satellites are visible and how they are distributed. The wider the satellite distribution and the more satellites are visible

the more accurate is the position information. The absolute position accuracy equals the dilution of precision times the measurement precision. So, if the measurement precision is 1m and the PDOP is 5, then the best position accuracy we can hope for is 5m. The maps in Switzerland are subdivided by a grid with cells of 1 $km^2$ each. I used that grid to estimate the accuracy of the device in the following way: First I located a number of points on the map then I drove there and sampled the latitude and longitude values at these points. The next step was to determine the mean values of the samples and transform them into Swiss coordinates according to [8]. Now I could calculate the deviation (see Figure 2.2). These figures are not very precise though in fact it is not much more accurate than 5m. An overlay of a sampled route and the map shows a similar picture (see Figure 2.3).

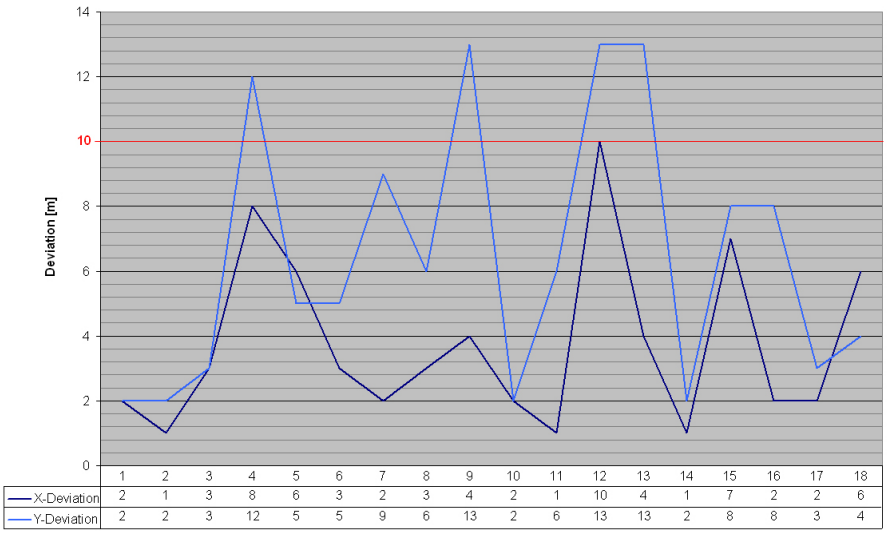


Figure 2.2: GPS position deviation from map coordinates

### Height

In order to determine how accurate the height data is, I decided to visit points that are marked on the map with exact heights. I used the *Landeskarte der Schweiz Blatt 1112 Ausgabe 1998, 1:25000* for that purpose. The height indicated by the GPS module does not correspond with the height marked on the swiss map. It has to be transformed first according to [8].

The deviation in Figure 2.4 is within a range of 11m. Even if more than four satellites are available the accuracy does not get much better. So the height is the least accurate information. In tendency the GPS values are a few meters above the map values.

### Speed

To measure the speed I went to a long straight part of a motorway. There I took the distance information that is marked every 200m along the road and determined the distance I drove and measured the time needed for that. With that information

I could calculate the approximate speed. Then I compared this speed to the speed of the car I was using, taking into account the error of the car's speed indicator which I got from a car magazine. The combination of these two methods allowed me to determine my actual speed at an accuracy of about 1km/h. It was not possible though to verify the specified 0.36km/h accuracy but I could verify that it is within 1km/h. Presumably this is enough for all practical purposes.

### 2.2.2 Reliability: Evaluation in Everyday Situations

To determine the reliability of the device, I chose two typical situations for the employment of the GPS system. The first one of them was a road trip that started at the harbor, led me through some smaller villages and over interurban roads. The second one was a walk through a canyon in the forest. As a measure I used the number of available satellites as well as the DOP (see figures 2.5-2.7). The minimum required number of satellites to determine the current position is three (four for the additional height information). A direct line of sight to the satellite is required.

#### Road

On the road I almost never had less than three satellites and the average number was around 6. It attracts attention that there are some peak values in the DOP-curve. They correspond always to the loss of at least one satellite. The opposite direction is not necessarily true though. The figures in the city (Zurich) were similar. I therefore do not present them here.

#### Forest

The walk through the forest shows a similar picture (Figure 2.4). I started the sampling outside the forest, walked about 15min through a small canyon covered with trees and ended up on a hill above it. The average number of satellites used was around 4—still enough to calculate the current position. On my way back down I ran (Figure 2.7). In this third chart it is remarkable that the average DOP value is much higher than in the other ones. It consists almost only of peaks although the average number of satellites was not much lower (around 3.5). The reason for the peaks is that we are close to the critical amount of the necessary three or four satellites, respectively.


Figure 2.3: Overlay of a sampled route and the map

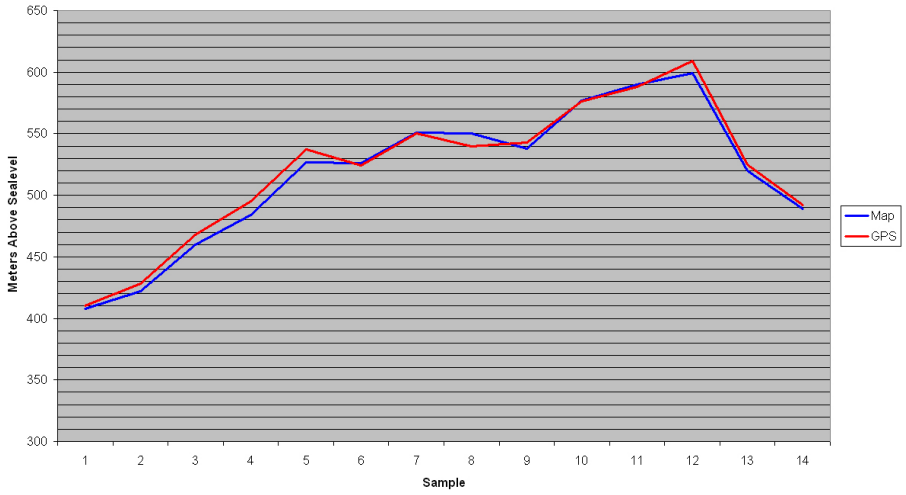


Figure 2.4: Height correlation between map and GPS

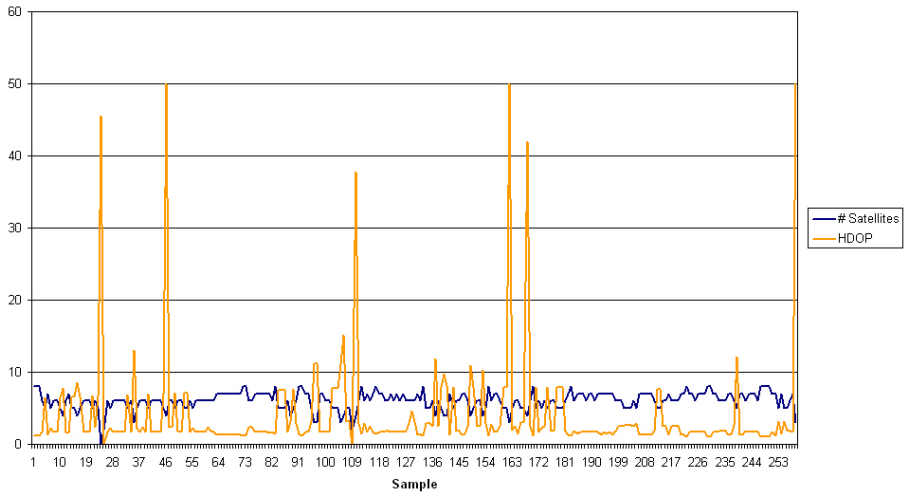


Figure 2.5: Road trip through villages

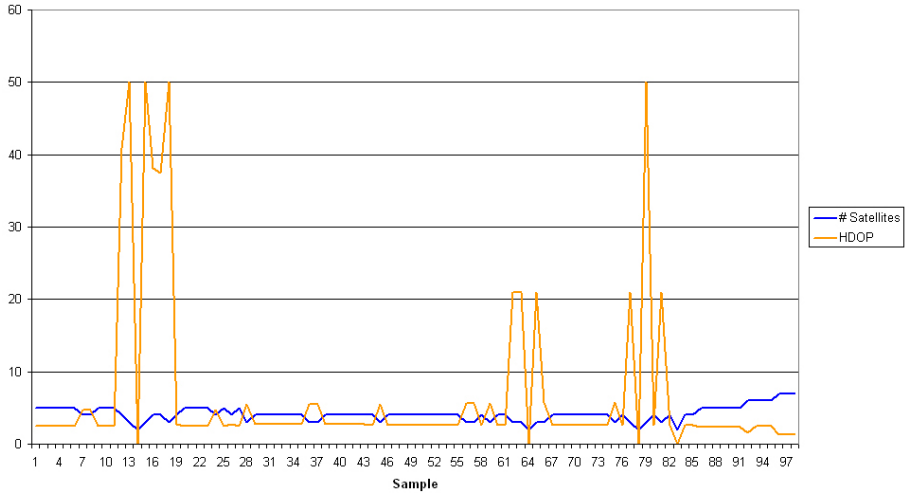


Figure 2.6: Walking through the forest

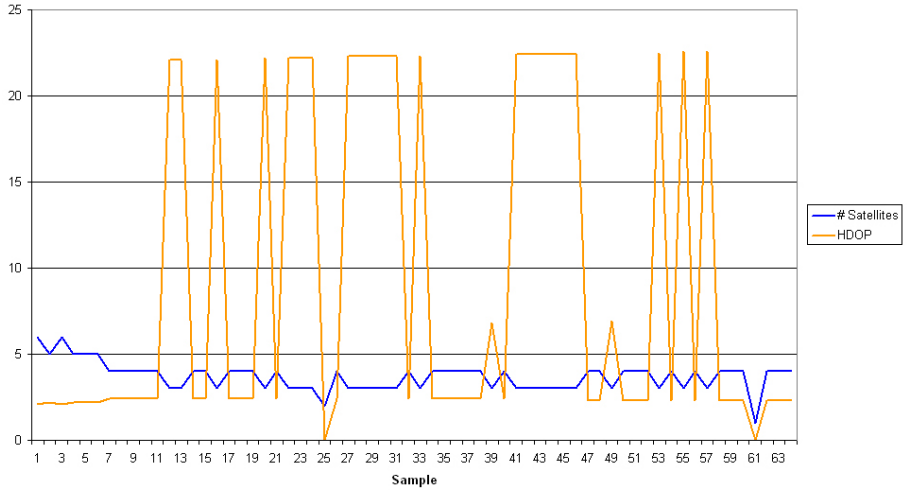


Figure 2.7: Running back through the forest

# Chapter 3

# The Java Programs

## 3.1 GPS API

The API that parses the NMEA0183 sentences mentioned in 2.1.1 was the core part of my thesis. First of all I had to get the data through the serial port of the iPAQ from the GPS receiver into my Java program. The corresponding API (Java Communications API) was defined but not implemented for the PocketPC. So I used an implementation written by James Nord [10] in order to communicate with the GPS receiver. The next step was to parse the NMEA0183 sentences. Each sentence begins with a '$' followed by some numbers separated by commas and ends with '<CR><LF>'. For the exact meanings of the numbers in the different sentences see the User Manual (Chapter 'Output Protocol') at [7].
I decided to use for each sentence an individual class in Java to store the current GPS information. Here is an example of how to use the GPS API:

```
1:   GPS gps = new GPS( "COM5:" );
2:   RMC rmc = new RMC();
3:   try {
4:      gps.updateRMC( rmc );
5:   } catch ( Exception e ) {}
```

Line `1` initializes the GPS API on *COM5*, Line `2` creates a new *RMC object* and Line `4` updates the RMC Object with the current GPS values. It is not possible to open the GPS API more than once otherwise an Exception will be thrown. All the information included in an RMC sentence can be accessed as member variables of the rmc object. In order to get the current latitude: `rmc.lat;` would for instance be used. All the Java classes contain exactly the same information as the corresponding NMEA0183 sentences. The only exception is the RMC class, where I have added a function `getSpeedKMH()` to get the current speed in terms of km/h instead of knots.
It is also possible to get the unparsed `InputStream` handle for the byte stream of the GPS receiver from the GPS API. For that purpose I have provided the following two functions:

```
public InputStream startGPS()
public void stopGPS()
```

In addition to this it is possible to flush a certain amount of characters onto the standard output with the function `flushData(int)`. The Javadoc to this API is available at [1].

### 3.1.1 GPS Tools

Some tools that might be useful but do not directly belong to the GPS API can be found in the class `GPS_Tools`. It includes `public Sats[] getAllSatsFromGSV()` which provides complete information about the visible satellites in a structured way, `public String nmeaLines(int)` which returns the specified number of NMEA0183 sentences as a string and finally a method to find out which ports are available in the system called `public String listAvailablePorts()`.

## 3.2 Java Applications

A further task of my thesis was to design and implement a sample application that is based on the API written in the first part. Instead of one large applications I wrote a few smaller ones that use various kinds of information provided by the GPS system. I decided to do the GUI with Thinlet [11], as it is relatively easy to learn and based on pure AWT, which is fast enough for the PocketPC. One of the main advantages of Thinlet is that you can describe your GUI in XML and without having knowledge about AWT, although it might be possible that there is a more efficient way to implement the GUI part directly with AWT. It would be possible to use Swing as well but referring to [9] it seems to be too slow and you need some tricks to get it running.
I implemented three programs designed for the iPAQ and one to visualize the stored GPS information on a desktop computer (see sections below). All of them operate based on the model-view-controller pattern and are fully multi-threaded. There is always a class containing the `main()` function which is responsible for showing the GUI and starting/stopping the threads. Their job is to keep updating the GUI with current information.

### 3.2.1 GPS Tracer

This application allows the user to see the GPS information in a human readable version. It consists of a few tabs containing various kinds of information about the current position (Figure 3.1), the satellites that are currently used (Figure 3.2), the dilution of precision, the tracking mode etc. Furthermore it is possible to watch the NMEA0183 sentences (unaltered) as soon as they are received (Figure 3.3). A last part provides the possibility to do a tracing over a time period. That allows the user to store all the gathered position samples into an XML file. The user can choose what should be stored (limited to the information contained in the GGA sentence) and in what interval the samples should be taken (Figure 3.4). This interval is limited to 2 seconds because it takes about 1 second to parse the appropriate sentence and the sentence is only received once a second). While tracing it is possible to watch the change of the altitude in an online chart (Figure 3.5).

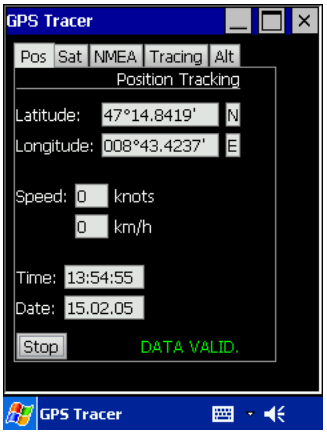


Figure 3.1: Current position information.

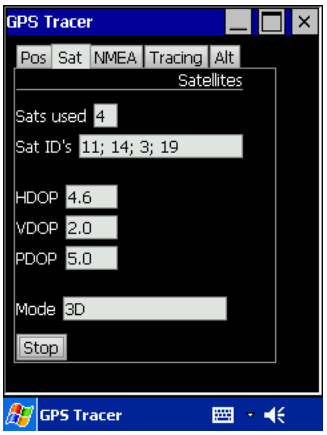


Figure 3.2: Currently used satellites.

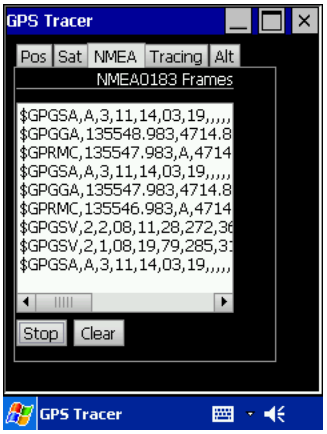


Figure 3.3: NMEA sentences.

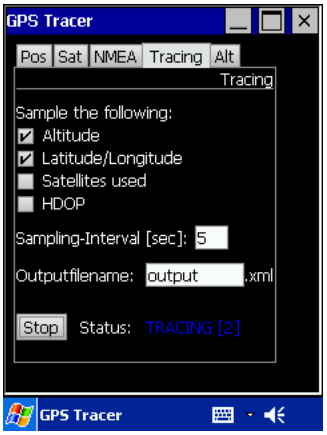


Figure 3.4: Recording GPS information.

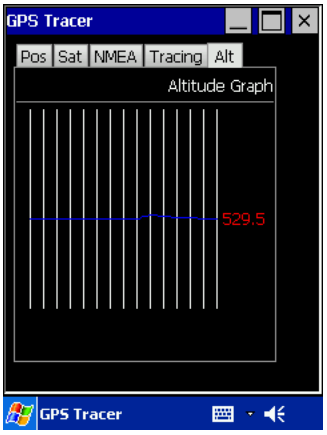


Figure 3.5: Altitude graph.


Figure 3.6: Painted route.

### 3.2.2 GPS Painter

The GPS painter was originally called "Human Pen". The idea is that the user becomes a pen on the world and the application paints his movements in real-time onto the iPAQ display (Figure 3.6).
Once the user has finished his/her journey it can be saved into an XML file similar to the one from the *GPS Tracer* (Figure 3.7). There are only minor differences between the two XML files.
To view the trace sometime later this XML file can be imported again; the program can be run in *Simulator mode* so it will not use the real GPS data but the ones from the XML file specified (Figure 3.8).
It is possible to set some options that will improve the way the current trace is displayed (Figure 3.8). These options include a zoom factor, the origin on the screen, the sampling interval and some visual options (draw sample points and connections between them). The zoom factor goes from 1% to 100% where the first reveals more details and the latter gives an overview of the journey. When 1% is used, one pixel corresponds approximately to one meter where with 100%, one pixel corresponds to one kilometer. These are just estimates to give an idea about the dimension. A summary of the current settings is available in the 'Info' menu.
The most difficult part of this application was to be able to paint directly on the GUI through its graphics context. I had to use a patched version of Thinlet because the original one did not support painting at that time. The patched version offers the possibility to define a `paint()` method that is called whenever it is needed. The method `repaint()` allows a manual invocation of the `paint()` function (see Chapter 5 for details).

### 3.2.3 GPS Visualizer

In order to display the gathered data (from the *GPS Tracer* as well as from the *GPS Painter*) on a desktop computer with larger capacity I wrote the *GPS Visualizer*. This tool has some additional features but is still based on the *GPS Painter*. For example specific colors for the background or the trace etc can be set or an image can be exported as a JPG or PNG file. This operation takes too long on the GPS

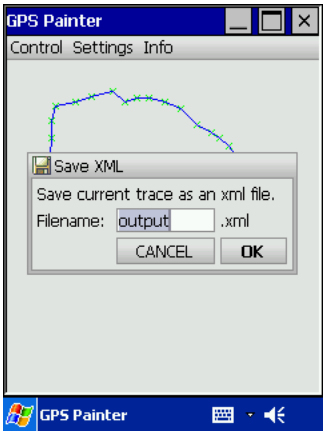


Figure 3.7: Save journey in XML.

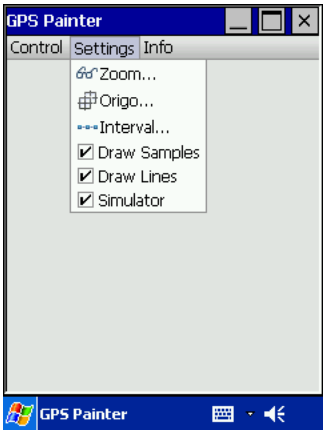


Figure 3.8: Available settings; activated simulator.


Figure 3.9: GPS Map.

iPAQ since Java 1.1 does not support these image export functions directly. I tried to solve this problem with a tool by Sun called *Jimi* [14] but the PocketPC was not fast enough. It took about one minute to produce the output of a trace with only a few dozen samples and the size of the image was restricted to about 800x800 pixels; I consequently decided to implement this functionality on the PC only.

### 3.2.4 GPS Map

What would GPS be without a map? So I implemented another variant of the *GPS Painter* this time with segments of the Swiss map in the background (Figure 3.9). The little red crosses indicate the visited positions. The map has been divided into segments of $1km^2$ each. If you leave one part of the map, the application switches to the next one and so forth. The trace will be saved so that if you return into a previous part your old position samples will be still there. It is also possible so switch between different map segments manually (Figure 3.10). In this tool the user is offered the possibility to save his journey into an XML file in the same way as in the *GPS Painter*. The accuracy is on average between five and ten meters. In the program there are currently $56km^2$ of the city of Zurich available but it is very easy to add more segments as long as they follow the predescribed rules. In order to work with the hard coded program settings the map segments must be parts of $1km^2$x$1km^2$ each and scaled to 234x234 pixels. The upper left corner of the square kilometer rectangle has to be 10 pixels off the top and left border.The square kilometer rectangle itself must be 200x200pixels. The code could also be easily adapted to meet different map specifications.

## 3.3 Work Progress and Difficulties

The first difficult thing was to find out how the Jeode JVM can communicate with the GPS receiver. I browsed the web and found the solution at http://www.teilo.net (see Chapter 5 for details). The parsing of the NMEA frames was not too tricky anymore after having read the exact specifications of the standard although it took me a while. Then I had to look for a possibility to design the GUI since I have never

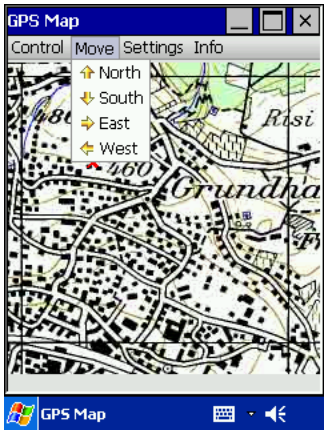


Figure 3.10: Switch between different map segments.

done that before. There are some valid alternatives to the pure AWT/SWT or Swing. I can recommend Thinlet since it is very well documented with some samples—a good start for anyone who needs a GUI and does not want to or cannot follow the common approach. The disadvantage of Thinlet showed up when it came to the real-time plot. I had to use a patched version of it in order to be able to draw into the GUI using its graphics context (see also Chapter 5).

# Chapter 4

# Conclusion

## 4.1 Jeode

One of the most popular advantages about Java is its platform independency. This thesis gave me a good opportunity to test it. Once I had written an application for Jeode (the Java Virtual Machine running on the iPAQ), there was no problem running the same application on the notebook as well. The other way round is not necessarly true though. On the laptop I wrote the applications for Java 1.4.2 whereas Jeode is based on the older Java 1.1 with a reduced function set. There are a lot of things not available on Java 1.1 when it comes to graphics. This increased the work a lot by things that should be available by default these days.

## 4.2 Thinlet

Thinlet offers a simple way to create nice looking AWT GUIs with almost every functionaliConclusionty. The idea of Thinlet is that the user specifies the look of the GUI in an XML file. This will then be parsed at runtime and the corresponding AWT GUI will be created.
I didn't miss anything except the possibility to paint directly in the GUIs graphics context. Fortunately this feature can be added by using the patched version of Thinlet (see Page 21). It should be available in the next standard version as well. Nevertheless I would rather use AWT directly for a next thesis. So you do not depend on a feature to be implemented but you are able to use the whole AWT API as such. It would be really bad if there was an important feature missing which was fortunately not the case during my work.

## 4.3 GPS

The GPS system needs a direct line of sight to the satellites in order to work. This results in some overhead during the testing phase. The programs have to be tested outdoors. It takes about one or two minutes every time the GPS has to reinitialize before it produces the first valid set of data. In order to circumvent this outside testing overhead, I wrote the simulator which is available in the *GPS Painter*.

# Chapter 5

# Useful Hints

As it was pretty tough to start in such an unfamiliar environment I would like to give some hints here how to do the most basic things with Java and the iPAQ PocketPC. Some very helpful tips and tricks concerning the installation of the JVM on the iPAQ, the basics of Jeode and debugging can be found in [9].

**Where to find some more information about Jeode.**

Some very useful information about Jeode can be found in the semester thesis by Nicolas Burri [9] and on the following website:
http://www.cs.unc.edu/ lindsey/7ds/notes/jeode

**How to set up Eclipse with the correct compiler for Jeode.**

Throughout the project I I worked with Java 1.1.8 as recommended in [9] and had no problems at all running the programs on the iPAQ. So download JDK 1.1.8 and unpack it somewhere into your system (e.g. C:\jdk1.1.8). For the next step start Eclipse and set the JRE as well as the corresponding compiler. Go to

`Window->Preferences->Java->Installed JREs`

and add the jdk1.1.8 (or whatever you are using) then go to `Compiler`, choose the tab `Compliance and Classfiles` and set the `Generated .class files compatibility` to 1.1.

**How to access a device through the serial port of the iPAQ.**

First of all you need to install the Java Communications API on the iPAQ as follows:

1. Download the archive from `http://www.teilo.net`.
2. Copy `CESerial.jar, comm.jar` and `javax.comm.properties` into a new folder

on the iPAQ (e.g. \windows\Java) and `commapi.dll` into \windows.
This driver for the serial port is free for non-commercial use.

Now you can access the serial port as described in [10]. There is some sample code which shows the recommended way of doing it.
Your link to start the java application on the iPAQ using the serial interface has to contain the Java Communications API in the classpath:

```
80#"\Windows\evm.exe" -cp \windows\Java\comm.jar;\windows\Java\CESerial.jar
MyClass
```

**Where to get the exact specifications for the NMEA0183 sentences from the HAiCOM GPS receiver.**

http://www.haicom.com.tw/support.htm

**How to test the GPS receiver.**

I used the little tool called "GPS Info" available at the HAiCOM website mentioned above.

**The serial port settings for the GPS receiver.**

Baudrate: 4800, Data bit: 8, Parity: None, Stop bit: 1, Flow control: None

**How to create Javadoc from source files.**

In your code you have to use the following syntax to create Javadoc contents in your source code:

```
/**
 * my Javadoc comment
 */
```

At the end you can export these comments as Javadoc by choosing

```
File->Export...->Javadoc
```

and specifying the source files you want to include.
For more details see [15].

### How to create a Thinlet GUI.

The idea behind Thinlet [11] is to specify the look and feel of an AWT GUI in an external file that is parsed whenever the program is executed. There are lots of easy-to-use components (called *widgets*) you can use to build up your GUI. In your Java code (the controller) you have to import that XML description and initialize a new Thinlet object (see example below). For details please refer to [11]. A very small example can be found at [12].

*Java example:*

```
1:   import thinlet.*;
2:   public class MyClass extends Thinlet {
4:     public MyClass() throws Exception {
5:       add(parse("description.xml"));
6:     }
7:     public static void main(String[] args) throws Exception {
8:       new FrameLauncher("MyClass", new MyClass(), 320, 240);
9:     }
10:  }
```

### I cannot find the file I have just created with my Java application?!

If you do not specify the whole path, the files are always stored in the root directory of the iPAQ.

### How to create image files with Java 1.1.

Since this feature is not yet supported you have to use something like Jimi [14]. It is very limited and slow due to the restricted resources of the iPAQ.

### How to paint into the graphics context of the Thinlet GUI.

This feature is not contained in the stable release yet. So you have to patch the thinlet.jar file.

1. Unpack the thinlet jar archive (`jar -xf thinlet.jar`).
2. Replace the file `Thinlet.java` with the one from [13] (rename it to Thinlet.java) and build the jar file again (`jar -cf thinlet.jar *`).[1]

Now you have an additional attribute for every component called `paint="myPaintMethod"` where you can specify a method that is called whenever a repaint of that specific component is needed. It needs to have the following signature: `public void`

[1] You can omit the MANIFEST-INF.

`myPaintMethod(Thinlet t, Object component, Graphics g, int width, int height)`. Therein you can redraw the component (e.g. with `g.drawLine(fromX, fromY, toX, toY)`).

**How to take screenshots on the iPAQ.**

There is a small freeware tool called *PocketSnap* available at [16]. On older devices it might be necessary to install the *.NET Compact Framework* first. It is free as well and available for download at [17].

# Bibliography


[1] "GPS on iPAQ" Website
http://bosco.ath.cx/gps

[2] U.S. Department of Defense Official Website
http://www.defenselink.mil

[3] GPS: A New Constellation - Exhibition Home Page
http://www.nasm.si.edu/exhibitions/gps

[4] Global Positioning System Overview
http://www.colorado.edu/geography/gcraft/notes/gps/gps_f.html

[5] The National Marine Electronics Association
http://www.nmea.org

[6] HAiCOM GPS
http://www.haicom.com.tw

[7] HAiCOM GPS - Support
http://www.haicom.com.tw/support.htm

[8] Formeln und Konstanten fuer die Berechnung der Schweizerischen schiefachsigen Zylinderprojektion und der Transformation zwischen Koordinatensystemen
http://www.swisstopo.ch/data/geo/refsysd.pdf (Page 11)

[9] Nicolas Burri: Semester Thesis "Java on the iPAQ", Summer 2003
http://dcg.ethz.ch/theses/ss03/ipaq_report.pdf

[10] Java Communications API Pocket PC/WinCE Driver
http://www.teilo.net/software/CEJavaComm

[11] Thinlet Website
http://www.thinlet.com

[12] Thinlet Tutorials
http://thinlet.blog-city.com/read/643909.htm

[13] Thinlet Paint Patch
http://uk.geocities.com/mike_hartshorn2/Thinlet_paintpatch.java

[14] JIMI Software Development Kit
http://java.sun.com/products/jimi

[15] How to Write Doc Comments for the Javadoc Tool
http://java.sun.com/j2se/javadoc/writingdoccomments

[16] nSonic Software - PocketSnap
http://nsonic.de/software/pocketpc/pocketsnap/pocketsnap.htm

[17] .Net Compact Framework
http://www.microsoft.com/downloads/details.aspx?FamilyID=359ea6da-fc5d-41cc-ac04-7bb50a134556&displaylang=en

# Appendix A

# Contents of the CD-ROM

### Additional Software

This folder contains a copy of the *Jeode JVM*, a copy of the *Java Communications API* and of the patched *Thinlet library*. These tools have to be installed prior to using the GPS software.

### GPS API

The core part of the software described in Chapter 3.1

### GPS Javadoc

The Java documentation for the GPS API.

### GPS Map

Application described in Chapter 3.2.4.

### GPS Painter

Application described in Chapter 3.2.2

### GPS SwissParser

A little tool that produces a summary of gathered GPS data (see README file for details).

### GPS Tracer

Application described in Chapter 3.2.1

### GPS Visualizer

Application described in Chapter 3.2.3

**INSTALL.txt**

This file contains installation instructions for all the software provided.

There is a README in each application directory with a closer description.