Introduction to GPS, Part 2

Presented by developerWorks, your source for great tutorials

ibm.com/developerWorks

Table of Contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

Section 1. Introduction

Should I take this tutorial?

This tutorial is designed for developers and product managers considering the Java programming language as an option for use in Global Positioning System (GPS) applications. This tutorial builds upon introductory material presented in the prior, prerequisite tutorial *Introduction to GPS, Part 1*. Developers with an interest in core Java I/O capabilities beyond the simple reading of text will learn how to communicate with a GPS device using byte-oriented data packets.

Additionally, this tutorial is a prerequisite for an upcoming tutorial discussing the opportunities and challenges associated with Java applications on PDA platforms.

What is this tutorial about?

This tutorial introduces the important Global Positioning System elements of *waypoints*, *routes*, and *tracks*, which constitute the majority of GPS applications; most data processing events in the GPS domain are related to one or more waypoints, routes, or tracks. This tutorial demonstrates interaction with a GPS device for the purpose of manipulating and capturing product data, date and time, and waypoints. This interaction is provided in the sample application, `ibmdwgps`, which is an extension of the application built in the prior tutorial (see the Sample application on page 20 section). From the Java perspective, this tutorial examines the topic of Java I/O as it creates a unique set of methods to support data acquisition from a GPS device. These methods implement a custom, layered protocol found in the Garmin family of GPS devices. Once this communications capability is established, the actual functionality is added to the sample application via additional menu selections. The tutorial consists of the following sections:

- GPS navigation and data on page 6 : This section discusses the basics of GPS positioning, GPS data elements, and popular protocols.
- Premium GPS data on page 8 : This section examines the combination and use of GPS data into useful applications, both in the field and in the lab.
- The Garmin Interface on page 11 : This section introduces the communications protocols employed by the Garmin family of GPS devices.
- Java and Garmin on page 15 : This section explores some of the challenges a Java developer faces in working with the Garmin interface protocol, which is designed around the C programming language. These challenges are examined and discussed, including a comparison of the two languages' support for structures over a stream.
- Sample application on page 20 : This section combines the user interface from *Introduction to GPS, Part 1* with the additional capabilities created with the Garmin interface methods.

Source code roadmap

This tutorial's sample application demonstrates an implementation of the Garmin Interface specification. Examples of C structure handling, Java data type parsing, and higher-level packet exchange logic are present. Additionally, there are examples of Java AWT and `javax.comm` communications programming. The following are snippets found in this tutorial.

***Note: It is highly recommended that you download the complete source code for the sample application before beginning this tutorial. See Tools on page 4 for a download link.**

- Link *Level Packet Definition.* This table breaks out the structure of the basic data structure exchanged by the Garmin Interface.
- *"C" structures and pointers.* This code snippet demonstrates the manner in which Garmin Interface packets are handled in the C language. The Garmin Interface is designed for the C programming language.
- *The Java byte array.* This snippet demonstrates building a communications packet with a Java byte array and transmission via an output stream.
- *Java Signed Bytes and the `adjustbyte` method.* This snippet describes the peculiar signed byte of the Java language and presents a mechanism for working with these bytes in the manner required for use in the Garmin Interface methods.
- `testbytes.class` : This snippet demonstrates handling multibyte data types between the Garmin Interface and Java. Included is a detailed explanation of byte ordering and bit shifting.
- `GetProductData`. This method initiates a request to a Garmin device, asking for product information.
- `GetDateTime`. This method initiates a request to a Garmin device, asking for date and time information.
- `GetWaypoints`. This method initiates a request to a Garmin device, asking for any available user waypoints.
- *Product Info Response.* This code demonstrates parsing a product info response packet, including multibyte data and string manipulation.
- *Date and Time Response.* This code snippet demonstrates multibyte handling to parse out the date and time sent from the unit.
- *Waypoint Response.* This code demonstrates the multipacket handling requirements of waypoint transfer.
- *Ack & Nak.* These methods send the appropriate response from the host to the Garmin unit.

Some helpful terms

- **E911.** An initiative to include emergency GPS locator capability into cell phones.

- **Byte stream.** Data sent from one device to another can be characterized as a stream of individual bytes.
- **Stream class.** Family of input/output classes available in Java code for the purposes of transporting information.
- **Two's Complement.** A technique used to represent numbers inside of a computer.
- **Triangulation.** Mechanism to ascertain exact location; i.e., a location in three dimensions provides comprehensive positioning information.
- **Almanac.** Database stored in GPS satellites containing current and anticipated position information.
- **Bearing.** The direction from the current location to a specific destination.
- **Elevation.** The altitude above sea level, usually measured in feet.
- **Heading.** Current direction of motion.

Tools

This tutorial builds upon the foundation of *Introduction to GPS, Part 1*, so the resources required are familiar. A key ingredient is Sun Microsystems's Java Communications API package for serial communications. The sample application is also designed to be "portable," therefore Java's Abstract Windowing Toolkit (AWT) is employed for the user interface. Also useful is a good reference to the Garmin Interface specification.

- Java SDK (JDK): The Java SDK is the core Java development platform required for any Java application. This must be installed to build the sample application. You can find the JDK at: *http://java.sun.com/j2se*.
- Java Communications API: You can find the Java package, `javax.comm`, used in the sample application for communicating with the GPS unit, at: *http://java.sun.com/products/javacomm/index.html*. Follow the steps outlined in the *Build environment* section of Part 1 of the GPS series to ensure proper installation of this package.
- Tutorial sample code: You can get the complete code to this tutorial's sample application at: *http://www.palm-communications.com/ibmdw* .
- GPS Unit: The tutorial's sample application demonstrates communications with a consumer GPS unit available from: *http://www.garmin.com/mobile/*. The unit must support the Garmin Interface protocol.
- Garmin's communications interface protocol documentation may be found at: *http://www.garmin.com/support/commProtocol.html*.

About the author

After his college basketball career came to an end without a multiyear contract to play for the L.A. Lakers, Frank Ableson shifted his focus to computer software design. He enjoys solving complex problems, particularly in the areas of communications and hardware interfacing. When not working, he can be found spending time with his wife

Nikki and their children. You can reach Frank at *frank@cfgsolutions.com*.

Section 2. GPS navigation and data

Global Positioning System: An application opportunity

The Global Positioning System is a collaborative network of space vehicles and land-based stations providing precise positioning information, assisting both man and machine to locate any point on Earth. While this system is organized and maintained by the U.S. Government, an entire industry exists around this network of GPS technology. There are applications providing aviation, maritime, and terrestrial customers with positioning data and services for commercial, military, and recreational use. As this technology becomes mainstream in society, the demand for additional functionality in the form of new applications increases. A market full of opportunity exists for innovative uses of positional information and services. Taking action on this opportunity requires an understanding of the GPS technology and the complementary technologies and tools necessary to harness GPS information. These knowledge areas include:

- Navigation techniques: How to get there from here.
- Available navigation data: What data is available to help?
- Data communication protocols: How is this data made available to applications?
- Programming languages, tools, and design patterns: Which languages and tools to use.
- Terrorism, health, safety, and logistics: Demand for emergency locator services are gaining ground, but is privacy at risk?

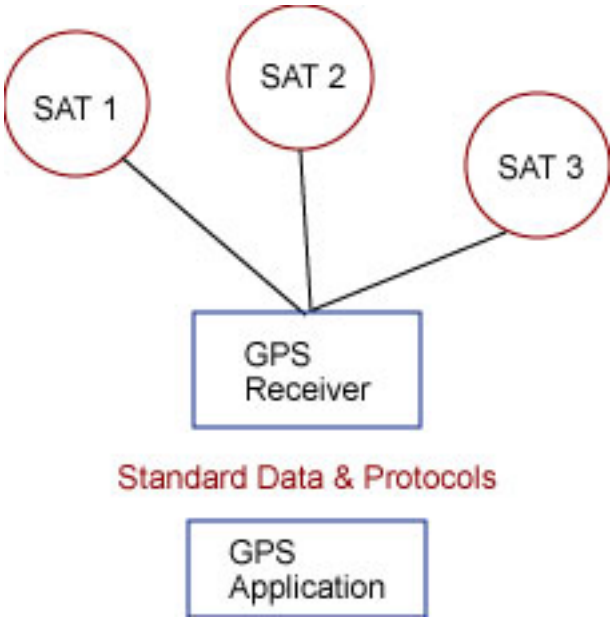
This tutorial examines some of the technical aspects of interacting with a GPS device for data capture purposes. To begin, the next panel reviews basic GPS operations.

Data protocols and formats

It is the responsibility of a GPS receiver to coalesce raw data received from multiple GPS satellites into accurate positional information. It is the job of applications to present this information to a user through a graphical user interface (GUI), or to feed this information to a number-crunching algorithm for further refinement and auto-navigation applications.

A time-tested design approach calls for modularity, meaning that each distinct function or activity is self-contained, allowing for change and enhancement over time without the need to change every portion of the product or custom solution. An essential characteristic of a successful modular design approach is a clear standard of communications between modules or layers. In the GPS domain, there are a few common formats, including the popular National Marine Electronics Association (NMEA). The first tutorial in this series examined the NMEA protocol. Additionally, GPS manufacturers define proprietary interfaces for communicating with their own

equipment. The following diagram depicts a layered design approach to GPS.



The most common consumer applications of GPS integrate road maps with GPS data. In this model, data taken from a GPS receiver indicating position is used to index into a database of highway maps, allowing the user to view their location on a map in real-time. The next section extends this discussion by examining specific forms of GPS data used in navigation applications.

Section 3. Premium GPS data

Waypoints

The fundamental element of GPS data is the *waypoint*. A waypoint is typically recorded in the field at a particular location of interest. For example, a prospector who stumbled upon a bounty of gold would mark the waypoint for subsequent use. The waypoint represents a specific location including:

Location name or description:

- Jackpot!

Location coordinates:

- N 40°55.418'
- W 074°42.864'

Elevation

- 800 ft

Bearing

- West

Some commercial GPS offerings include collections known as *points of interest*. Points of interest are essentially categorized lists of waypoints. For example, lists of cities, restaurants, airports, or campgrounds are available for popular GPS devices. User waypoints can also be stored and categorized. Waypoint management and presentation is an opportunity area for development of new GPS applications.

The next panel discusses routes -- the linking of multiple waypoints into a planned course.

Routes

A *route* is a collection of intermediate waypoints leading to a chosen location or terminus. Waypoints can be used to break the trip into smaller segments, or to simply ensure that every point of interest along the way is visited. A route can include any combination of waypoints, points of interest, or simply specific coordinates. In fact, a scavenger hunt based on a route of only coordinates might make for good fun! Here is a hypothetical route marking waypoints along the way from New York to Washington DC, including some important stops along the way:

Waypoint	Distance
Starting Point	N/A
Point of Interest 1; Yankee Stadium	22 miles
Point of Interest 2; Restaurant, Taco Bell	4.5 miles
Point of Interest 3; Train Station	.9 miles
Point of Interest 4; Washington D.C.	242 miles

Embedded software running in GPS devices allows editing of routes, including searching on device lists of waypoints and maps. Richer functionality is available via mapping software for desktop computers. These desktop applications integrate many points of interest and provide enhanced color graphics and mapping capabilities. Once the routes are established on the desktop, they can be transferred to the GPS unit for use in the field. A typical GPS unit contains enough storage capacity to maintain a few dozen routes with many waypoints each.

While routes plan the trip, tracks record the path taken. The next panel examines tracks, which are the breadcrumbs of GPS.

Tracks

In defiance of the proverb "as the crow flies," for most people, the distance between two points is not a straight line. So, just how far is it from point A to point B? This distance can be measured with GPS tracks. GPS track data is an electronic recording of the path taken. As a course is followed, or as someone simply meanders about without a predetermined route, the GPS unit records time and position data. There are two primary uses of this information:

- It is used to record the distance traveled and the time elapsed during travel. This kind of information is quite useful for recreational and commercial applications alike. For example, a jogger can record the exact distance traveled, a task not always easy to accomplish, particularly for off-road courses. In a commercial application, a delivery vehicle in a city environment can accurately measure the distance between two city addresses given a myriad of One-Way signs and No Left Turn restrictions during business hours. The vehicle's odometer is capable of measuring distance traveled, however tracks provide a history of the actual course taken.
- The second use of GPS tracks is the proverbial breadcrumb application. Track data can be used to retrace a path to return to the point of departure. This is particularly useful when exploring an unfamiliar area while the options for conventional navigation are unavailable or impractical.

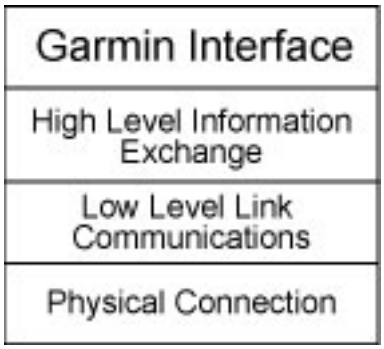
Tracks are stored in GPS units as a series of data points recording date, time, and position. These tracks can consume a rather large amount of memory, so GPS devices often allow a handful of tracks to be stored independently of one another, but there is a finite amount of storage space available for all track data.

The next section introduces a data and communications protocol for a popular line of GPS devices from Garmin.

Section 4. The Garmin Interface

Basic protocol design

The Garmin product family shares a common data and communications protocol, known as the Garmin Interface. This protocol provides a mechanism for all properly equipped Garmin hardware products to exchange data with another device. This other device is typically a personal computer or PDA. The protocols consist of three layers, again demonstrating the venerable modular design strategy:



The highest level in the stack, indicated as "High Level Information Exchange," encompasses a significant number of data flows including product requests, transfers of GPS data such as waypoints or routes and much more. It is up to the GPS application developer to extract (or load) information from (or to) the GPS unit and integrate that data with information from other applications. The form of data transferred via these protocols is often product specific; however, the protocol is generally applicable to the entire Garmin product line.

The next panel examines the Garmin Interface's lowest layer -- the physical layer.

Communications parameters

The Garmin Interface's lowest layer, the physical layer, calls for devices to communicate with the following communications parameters:

- 9600 Baud (minimum)
- 8 Data bits
- No Parity
- 1 Stop bit
- Full duplex

The Garmin Interface also requires a traditional UART communications interface. Fortunately, this type of interface is found on virtually every late model personal

computer and PDA. For more information on UART communications, please refer to Tutorial resources on page 30 . The cables available for Garmin products have a DB9 connector to conveniently connect to a personal computer. The Garmin device side of the cable, however, is a custom connector, specifically for use on the Garmin device.

The next few panels examine the techniques available for extracting data from a GPS unit by drilling down into various aspects of the Garmin Interface.

The data packets

Communication between a host, such as a PC or PDA, and a Garmin device, is accomplished through the exchange of data packets. Each data packet contains a sequence of bytes including delimiters, command instructions, payload data, and a checksum for error handling. An acknowledgement must be sent back to the sender for each received data packet. The acknowledgement is either an affirmative response, known as an *ACK*, or a negative acknowledgement, known simply as *NAK*. Here is the low-level packet format:

Packet Byte Offset	**Description**	**Comments**
0	0x10 character	This is the very first byte in the packet and serves as a delimiter for the packet. This byte is commonly referred to as Data Link Escape, or *DLE*.
1	Packet Identifier, or Command Byte	This byte identifies the packet, indicating the type of command this packet represents. A list of common packet types is presented in the next panel.
2	Size of packet in bytes	This byte holds the size of the packet payload.
3	Packet payload	Up to 255 bytes of data.
Size of packet -- 3	Checksum	This byte value is the "2s complement" of the sum of all bytes starting from the packet identifier and including all of the packet data.
Size of packet -- 2	0x10	This is the second-to-last byte and serves as a message delimiter. Again, this is the DLE character.
Size of packet -- 1	0x03	This byte marks the end of the text. This is commonly referred to as ETX.

Care must be taken when constructing packets; it is feasible to have the packet size value or checksum value be the same as the DLE character. This can cause the receiving device to misinterpret the incoming packet. This situation is resolved by simply transmitting an additional DLE character immediately after the non-delimiter DLE. The additional DLE bytes are not added to the size of the payload. Note that this scenario can occur also when a byte in the payload is equal to the DLE character. The

use of the ETX character as the end of text indicator prevents two DLE characters from being transmitted back-to-back in normal operation when sending successive packets:

```
<DLE><PID><SIZE><PACKET><CHECKSUM><DLE>
<ETX><DLE><PID><SIZE><PACKET><CHECKSUM>
<DLE><ETX>
```

The next panel introduces the packet types used in this tutorial.

Packet types

Link protocol packets come in all varieties. Beyond the ACK and NAK packets, the most common packet type is the *Product Request*. This packet is sent by a host application when initiating communications with a GPS device in an effort to identify the model and capabilities of the unit. The lowest level of the Garmin Interface protocol supports multiple packet types, accommodating many data communications scenarios. Some of the packet identifiers used in this tutorial include:

Packet Identifier (second byte in a packet, decimal values shown)	**Description**
6	Acknowledgement (ACK)
10	Command Data
12	Transfer complete
14	Date and Time Data
21	Negative Acknowledgement (NAK)
27	Records (indicates the number of records available for transfer)
35	Waypoint data
254	Product Request. This is generally the first packet sent during a communications session.
255	Product Data. This packet is sent in response to the Product Request packet. This is then followed by a "253" packet ID, which provides a capabilities list.

The next panel examines this process in more detail with the specific commands required for retrieving Product Information Data from a Garmin eTrex Venture unit.

Packet exchange: applications

Communicating with a Garmin GPS unit resembles a conversation that humans might

engage in; the session is essentially an exchange of sentences. In this case, it is an exchange of command and response packets. A common transaction between a host PC and Garmin GPS unit is the transfer of waypoints. Here are the steps involved, along with some specific Garmin packet identifiers. An underlying assumption is that all packets are cleanly transferred and no negative acknowledgements are required.

- The host issues the `Pid_Product_Rqst` packet. The GPS unit responds with a positive acknowledgement packet followed by a `Pid_Product_Data` packet. This packet contains the product ID, software revision and some textual descriptions of the product. This data can be used to further refine subsequent activities and queries. For example, each version of a GPS unit stores waypoints in a different manner, managing unique sets of attributes for each waypoint. By identifying the GPS model, the host can properly parse out any responses from the GPS unit. The host issues an acknowledgement packet, or `Pid_Ack_Byte` to the GPS unit. The GPS unit sends a `Pid_Protocol_Array` packet, indicating the capabilities of this device. The host acknowledges this packet as well.
- The host next requests waypoints from the GPS unit with the issuance of the `Pid_Command_Data` packet. The subcommand of this packet is set to 7, the command for waypoint transfer request.
- The GPS provides a `Pid_Ack_Byte`, followed by a `Pid_Records` packet. This packet informs the host of the number of waypoint records to expect. This is useful for the host application if it needs to allocate memory or persistent storage to retain the waypoints upon receipt. Also, the application can choose to provide a progress dialog, providing feedback to the user, informing the user of the overall progress of the transfer. The host acknowledges the `Pid_Records` packet, informing the GPS unit that it cany commence with the transfer of the first `Pid_Wpt_Data` packet.
- The host acknowledges each waypoint with the `Pid_Ack_Byte` Packet and then processes the newly received waypoint by displaying attributes or saving it in some manner.
- When the final waypoint has been acknowledged, the GPS unit sends a `Pid_Xfer_Cmplt` packet informing the host that the transfer is finished. The host acknowledges this packet and the session is complete.

The next section takes a closer look at the specific Garmin Interface packets and the issues associated with processing them in the Java language.

Section 5. Java and Garmin

The "C" problem

When a packet is received, and the checksum verified, the resulting packet payload data must be interpreted to obtain the useful contents. The data types exchanged in the Garmin protocol are designed for the C programming language, complex structures consisting of primitive data types. For example, the `Pid_Date_Time_Data` packet payload looks like this:

```
typedef struct
{
byte month; /* month (1-12) */
byte day; /* day (1-31) */
word year; /* year (1990 means 1990) */
int hour; /* hour (0-23) */
byte minute; /* minute (0-59) */
byte second; /* second (0-59) */
} D600_Date_Time_Type;
```

In the C language, a data structure like this is manipulated through the use of pointers. A `D600_Date_Time_Type` pointer is employed to "point to" the memory location holding the packet payload and then cause the program to "interpret" the next eight bytes as a `D600_Date_Time_Type`. Here is a representative code snippet employing the C language:

```
unsigned char buf[200];
D600_Data_Time_Type * pDateTime = NULL;
....
// packet payload is contained in buf, starting at offset 3 (actually fourth byte)
pDateTime = (D600_Data_Time_Type *) &buf[3];
printf("The month is [%d]\n",pDateTime->month);
```

This data type definition is taken from the Garmin Interface Specification document. See Tutorial resources on page 30 for more information on obtaining this document.

Data transfer requirements

Transferring data across a serial connection with the Garmin Interface requires that each data structure be *serialized* into a byte stream with the following characteristics:

- All structures are byte packed. This means that there are no padding bytes used. Padding bytes are commonly used for optimizing memory access, placing each member on an even byte boundary.
- Multibyte elements, such as integers, are transferred in little-endian format. This means that the first byte stored in memory is the lower order byte, followed by bytes of increasing order.

- Structure members are serialized in sequence.

The next panel introduces the byte data type in the Java language.

Byte handling in Java

Java classes performing communications functions often use the byte array data type. It conveniently and efficiently stores data, providing indexed access to any element of the array. Many I/O class methods employ byte arrays as parameters, simplifying such things as sending data. The methods interrogate the byte array and send the entire byte array, unless explicitly told otherwise. For example, the next code snippet demonstrates building a Garmin Interface `Pid_Product_Rqst` packet:

```
byte[] request = new byte[6];
request[0] = 0x10;            // DLE
request[1] = (byte) 0xfe;     // Packet Id
request[2] = 0x00;            // size of packet payload (there is none!)
request[3] = (byte) (0x100 - request[1]);           // checksum
request[4] = 0x10;            // DLE
request[5] = 0x03;            // ETX
try
{
os.write(request);
os.flush();
}
catch (Exception e)
{
...
}
```

The byte array request is constructed and passed to the write method of the OutputStream, os. This simple method invocation sends each byte in the array.

The next panel discusses some mathematical characteristics of the Java byte.

To sign or not to sign?

Java's handling of individual bytes is rather divergent from that of other languages. A byte in Java is a signed value. While a Java byte does contain eight bits, it does not hold values above 127; a byte value spans from -128 to 127. The C language byte, or unsigned char, holds values ranging from 0 to 255. Most byte-oriented communications protocols, including the Garmin Interface, define values as unsigned bytes. This represents a challenge for Garmin protocol handling in Java code. Of course, there is a solution.

When performing arithmetic, Java technology promotes bytes into integers. This fact allows a simple transformation to represent "unsigned" byte values in Java code. Here is a helper function defined in the sample application used to handle this characteristic

of Java signed bytes:

```
private int adjustbyte(byte b)
{
if (b < 0)
return 256 + b;
else
return b;
}
```

This routine compares the byte's value to determine if it is less than 0. If so, it adds 256 to it, resulting in an integer value that is capable of representing values larger than 127. If the value is greater or equal to 0, it simply returns the byte value. In either case, the return value is equivalent in value to an unsigned byte found in C.

The next panel demonstrates the steps required to parse byte arrays into Garmin Interface compatible values.

Java data-type parsing

While type casts exist in Java code, its approach to memory handling is considerably less maverick than that of the C language; Java language does not permit explicit pointers to memory locations. The net effect of this practice is the need to parse byte arrays into Java data types instead of simply *casting* or *promoting* them to structures, as in C. Handling the Date and Time packet presented earlier requires the ability to parse bytes, words, and ints. In the snippet below, a byte array named packet contains the raw data received from the GPS unit. Here is a snippet from the application for handling the Date and Time packet response:

```
int b;
trace.append("Handling packet type 14, Date & Time Information.\r\n");
i = 2;  // skip past packet id and packet size
// get month
b = adjustbyte(packet[i++]);
trace.append("Month is [" + b + "].\r\n");
```

Notice how the byte values are simply *adjusted* and easily handled. Also observe that the offset into the packet, variable `i`, is incremented as each field is handled.

The next panel demonstrates multibyte handling and contemplates the effect of byte order on this process.

Multibyte parsing

Multibyte values require more consideration than single bytes. The data used in the Garmin Interface protocol is represented in little-endian fashion. The following snippet demonstrates the construction of a 16-bit value from two bytes. To handle this data, the

`myint` variable is assigned the value of the adjusted first byte. Then, the adjusted second byte is shifted 8 bits and added to the first value. This is done because the second byte is actually the more significant byte in the little-endian architecture. Remember, the adjustment process is required because Java code does not permit the casual copying of memory locations. Even if it were to allow for memory copying or pointer use, the byte order issue would still need addressing.

```
public class testbytes
{
private int adjustbyte(byte b)
   {
if (b < 0)
    return 256 + b;
else
    return b;
   }
public static void main(String args[])
   {
testbytes t = new testbytes();
byte b1,b2;
int myint;
b1 = (byte) 0xd2;
b2 = (byte) 0x07;
myint = t.adjustbyte(b1);
myint = myint + (t.adjustbyte(b2) << 8);
System.out.println("myint is [" + myint + "]");
System.out.println("complete.");
   }
}
```

The innocuous looking values of 0xd2 and 7 actually combine to equal the value 2002. The next panel shows how it works.

Multibyte parsing (cont.)

Below is a demonstration of how the values 0xd2 and 7 actually combine to equal the value 2002.

- The first byte is taken at face value -- 0xd2 is equivalent to 210 decimal.
- The second byte, 7, is shifted 8 bits to the left. Here is a binary representation to make things clearer:

```
31
268421
73100052 1
68994215 2631
84268426 84268421
-------- --------
00000000 00000111
= 7
```

- Shifting the value 8 bits to the left results in this binary representation:

```
31
268421
73100052 1
68994215 2631
84268426 84268421
-------- --------
00000111 00000000
= 1792

210 + 1792 = 2002
```

As an interesting exercise, the value 2002 in hexadecimal is written as 7D2. However, in physical memory, and when serialized across a data stream in the Garmin Interface, the actual storage is d207.

This snippet is available as a Java source file named `testbytes.java` and is part of the source bundle available with this tutorial. The Tools on page 4 panel contains a link to the source code.

The next section begins the detailed examination of the sample application.

Section 6. Sample application

Motivation and purpose of sample

This tutorial's sample application builds upon the application built in *Introduction to GPS, Part 1*, `ibmdwgps`. This application adds new menu selections for processing specific Garmin Interface transactions. The support for NMEA sentence parsing and simple data logging are still supported, as is the ability to record sessions to a disk file. Once complete, the application has the following features:

The Tools on page 4 panel contains a link to the complete source code for this application.

The sample application was tested against a Garmin etrex Venture model handheld GPS unit communicating via the Garmin Interface protocol at 9600 baud.

The application is named `ibmdwgps`, just as in Part 1. This tutorial's version is referred to as version 2.0.

Configuration and protocol settings

An additional value is permitted in the properties file utilized for the application. In the previous application, the only acceptable value for the optional protocol property was *NMEA*. This tutorial allows for three possible interpretations of the data, controlled by the `ibmdwgps.properties` file or the command line. The following table summarizes the acceptable values for the class's operation.

Protocol Offset	Baud Rate	Comments
NMEA	4800	NMEA protocol support includes parsing of NMEA sentences, utilizing simple Java string manipulation techniques.
Garmin	9600	GARMIN protocol support enables additional menu selections. The program expects GARMIN packets and will not attempt to parse NMEA data.
Any other value, including omission altogether	ANY	In this mode, the application simply captures text and records the data to the scrolling text area. The baud rate must match the device's baud rate setting.

Here is a snapshot of the property file:

```
## ibmdwgps.properties
version=Version 2.0\r\n
welcomemessage=Welcome to IBM developerWorks : Wireless: Waypoints,
Routes, Tracks, and the Garmin Interface\r\n
commportnumber=COM1
baudrate=9600
protocol=GARMIN
```

When the application initializes, the protocol is loaded from the properties and class level flags are set to indicate which protocol to expect. The configuration reading portions of `ibmdwgps.java` have been modified to accommodate this new protocol option. The next few panels discuss each of the newly added Garmin Interface transactions.

User interface

`Ibmdwgps` version 2 introduces a new menu selection, named Garmin. Here is the code to add these menus:

```
private Menu mnuGarmin;
private MenuItem miGetProductData;
private MenuItem miGetDateTime;
private MenuItem miGetWayPoint;


mnuGarmin = new Menu("Garmin");

miGetProductData = new MenuItem("Get Product Data");
miGetProductData.addActionListener(this);
mnuGarmin.add(miGetProductData);

miGetDateTime = new MenuItem("Get Date/Time");
miGetDateTime.addActionListener(this);
mnuGarmin.add(miGetDateTime);

miGetWayPoint = new MenuItem("Get WayPoints");
miGetWayPoint.addActionListener(this);
mnuGarmin.add(miGetWayPoint);

barMain.add(mnuGarmin);
```

Action handler

The `ibmdwgps` class implements the ActionListener interface, allowing it to respond to user actions. In order to implement this interface, a class must provide a method named `actionPerformed`, taking a single Event parameter. Here is the action handler code showing the new menu options for the Garmin protocol:

```
public void actionPerformed(ActionEvent e)
```

```
{
   String cmd = e.getActionCommand();
   System.err.println(cmd);
...
   if (cmd.equals("Get Product Data"))
   {
      getProductData();
   }
   if (cmd.equals("Get Date/Time"))
   {
      getDateTime();
   }
   if (cmd.equals("Get WayPoints"))
   {
      getWaypoints();
   }
...
}
```

This function is invoked when the user selects a menu option. This method essentially acts as a distributor of events. For example, when the "Get Product Data" event is performed, this method transfers control to the `getProductData` method. The next panel examines the `getProductData` method.

Get product data option

Requesting product data from the GPS unit requires that the application implement the Garmin protocol interface, including the proper sequence of requests and acknowledgements. The method `getProductData` initiates a request for product information. Here is the code:

```
public void getProductData()
{
   byte[] request = new byte[6];
   if (!portisopen)
   {
      trace.append("Please select Start under the Options menu
          to open the port first!\r\n");
      return;
   }
   trace.append("Sending request to GPS unit.\r\n");
   request[0] = 0x10;            // DLE
   request[1] = (byte) 0xfe;     // Packet Id
   request[2] = 0x00;            // size of packet payload (there is none!)
   request[3] = (byte) (0x100 - request[1]);           // checksum
   request[4] = 0x10;            // DLE
   request[5] = 0x03;            // ETX
   try
   {
      os.write(request);
      os.flush();
   }
   catch (Exception e)
   {
      trace.append("Error requesting product information
          [" + e.getMessage() + "]\r\n");
```

```
      e.printStackTrace();
   }
   trace.append("Request sent.\r\n");
}
```

Things to note in this packet creation:

- DLE bytes delimit the message
- The packet identifier is 0xfe (254 decimal), indicating that this is a `Pid_Product_Rqst` packet. Note that this value requires a cast to a Java byte. Without this cast, the Java compiler warns of a possible loss of precision.
- There is no payload data for this packet, so the size of the payload is set to zero.
- The checksum calculation is simply 256 -- 254, the packet id. The checksum equates to 2, which is the two's complement of 254. For more information on two's complement, please see Tutorial resources on page 30 . Note that the hexadecimal value 0x100 used in the code is equivalent to 256 decimal.

The next panel presents the code necessary for initiating a Date and Time request.

Get Date and Time info

One of the most important pieces of information a GPS receiver can offer beyond position data is accurate date and time information. Luckily, this is easily achieved with a query to the Garmin device. The mechanism used is the `Pid_Command_Data` packet. This packet sends additional payload bytes, indicating the specific command required. Here is the code:

```
public void getDateTime()
{
   byte[] request = new byte[8];
   if (!portisopen)
   {
      trace.append("Please select Start under the Options menu
          to open the port first!\r\n");
      return;
   }
   trace.append("Sending request to GPS unit.\r\n");
   request[0] = 0x10;            // DLE
   request[1] = 0x0A;            // Packet Id
   request[2] = 0x02;            // size of packet payload
   request[3] = 0x05;            // command byte 5, transfer time
   request[4] = 0x00;
   request[5] = (byte) (0x100 - (request[1] + request[2] +
                 request[3] + request[4]));
// checksum
   request[6] = 0x10;            // DLE
   request[7] = 0x03;            // ETX
   try
   {
      os.write(request);
      os.flush();
   }
   catch (Exception e)
```

```
   {
      trace.append("Error requesting date & time information
               [" + e.getMessage() + "]\r\n");
      e.printStackTrace();
   }
   trace.append("Request sent.\r\n");
}
```

Things to note in this code:

- The packet identifier is 0x0A (decimal 10), indicating that this is a `Pid_Command_Data` packet. Note that this value requires a cast to a Java byte. Without this cast, the Java compiler warns of a possible loss of precision.
- The payload for this packet is an integer value of 5. Note that this value is actually a 16-bit value, spread over two consecutive bytes, in little-endian format. The value 5 corresponds to the `Cmnd_Transfer_Time` sub command.
- The checksum calculation is simply 256 -- the sum of bytes sent, starting with the packet identifier.

The next panel presents the steps necessary to initiate a waypoint transfer.

Get waypoint data

As mentioned earlier, the waypoint represents key GPS data, indicating a position and heading. The process of retrieving waypoints from a Garmin unit goes beyond the steps necessary in the previous examples of Product and Date and Time data retrieval. As discusses previously, waypoint transfer involves a dialog for each and every waypoint record transferred. Here is the Transfer Waypoint request code:

```
public void getWaypoints()
{
   byte[] request = new byte[8];
   if (!portisopen)
   {
      trace.append("Please select Start under the Options menu
               to open the port first!\r\n");
      return;
   }
   trace.append("Sending request to GPS unit.\r\n");
   request[0] = 0x10;            // DLE
   request[1] = 0x0A;            // Packet Id
   request[2] = 0x02;            // size of packet payload
   request[3] = 0x07;            // command byte 7, transfer waypoints
   request[4] = 0x00;
   request[5] = (byte) (0x100 - (request[1] + request[2] +
                request[3] + request[4]));
// checksum
   request[6] = 0x10;            // DLE
   request[7] = 0x03;            // ETX
   try
   {
      os.write(request);
      os.flush();
```

```
   }
   catch (Exception e)
   {
      trace.append("Error requesting waypoints information
                  [" + e.getMessage() + "]\r\n");
      e.printStackTrace();
   }
   trace.append("Request sent.\r\n");
}
```

The payload for this packet is an integer value of 7. Note that this value is actually a 16-bit value, spread over two consecutive bytes, in little-endian format. The value 7 corresponds to the `Cmnd_Transfer_Wpt` sub command.

Most of the work done for waypoint transfers is actually accomplished in the `processpacket` method, introduced in the next panel.

Packet handling

Data received from the GPS unit is processed a packet at a time by the processpacket method. This method must do a number of things:

- Verify the checksum value
- Identify the packet type.
- Provide acknowledgements, as required
- Parse incoming packet payload

The code is a `switch` statement, with a case to handle every expected packet type. This code is part of the source code available from Tools on page 4 .

Subsequent panels discuss specific packet handling logic.

Product information response

In response to the request for Product information, the GPS unit sends a `Pid_Product_data` packet. It is parsed out byte by byte in the following snippet:

```
...
case 255:
   trace.append("Product Data Response Data Packet\r\n");
   i = 2;  // skip past packet id and packet size
   // a 16 bit integer representing product ID
   // first byte is lower order
   myint = (int) adjustbyte(packet[i++]);
   myint = myint + (adjustbyte(packet[i++]) << 8);
   trace.append("\tProduct ID is [" + myint + "]\r\n");
   // a 16 bit integer representing software version
```

```
   myint = (int) adjustbyte(packet[i++]);
   myint = myint + (adjustbyte(packet[i++]) << 8);
   fsoftwarerev = (double) myint / 100;
   trace.append("\tSoftware Revision is [" + fsoftwarerev + "]\r\n");
   // multiple null terminated strings....
   while (i < packet.length-1)
   {
      if (packet[i] != 0x00)
      {
         buf.append((char) packet[i]);
      }
      else
      {
         trace.append("\t" + new String(buf) + "\r\n");
         buf = new StringBuffer();
      }
      i++;
   }
   ackpacket((byte)255);
   break;
...
```

Things to note in this snippet:

- Use of the `adjustbyte` method to "fix" the unsigned/signed byte differences discussed earlier.
- Incrementing of the index variable, `i`, to walk through the byte array.
- Multibyte processing with the bit shift mechanism demonstrated earlier in the `testbytes.class` example.
- The acquisition of an arbitrary number of character strings, each delimited with a zero byte.
- Use of the `ackPacket` method, used to positively acknowledge this Product Data packet response, packet id 255.

The next panel examines the response containing Date and Time information.

Date and Time response

In response to the request for Date and Time information, the GPS unit sends a `Pid_Date_Time_Data` packet. It is parsed out byte by byte in the following snippet:

```
...
case 14:
   trace.append("Handling packet type 14, Date & Time Information.\r\n");
   i = 2;  // skip past packet id and packet size
   // get month
   b = (byte) adjustbyte(packet[i++]);
   trace.append("Month is [" + b + "].\r\n");
   b = (byte) adjustbyte(packet[i++]);
   trace.append("Day is [" + b + "].\r\n");
   myint = adjustbyte(packet[i++]);
   myint = myint + (adjustbyte(packet[i++]) << 8);
   trace.append("Year is [" + myint + "].\r\n");
```

```
   myint = adjustbyte(packet[i++]);
   myint = myint + (adjustbyte(packet[i++]) << 8);
   trace.append("Hour is [" + myint + "].\r\n");
   b = (byte) adjustbyte(packet[i++]);
   trace.append("Minute is [" + b + "].\r\n");
   b = (byte) adjustbyte(packet[i++]);
   trace.append("Second is [" + b + "].\r\n");
   ackpacket((byte) 14);
   break;
...
```

Note that the `ackPacket` method is used to positively acknowledge this Date and Time packet response, packet id 14.

The next panel examines the response to the request for waypoints.

Waypoint response

There are actually three distinct packet types that comprise the response of the Transfer Waypoints request. The first response is a `Pid_Records` packet:

```
...
case 27:
   trace.append("Handling Records data type.\r\n");
   // record the number of records available
   // (if we were allocating memory to hold these, for example ..)
   i = 2;  // skip past packet id and packet size
   myint = adjustbyte(packet[i++]);
   myint = myint + (adjustbyte(packet[i++]) << 8);
   trace.append("There are [" + myint + "] records available.\r\n");
   // let's acknowledge this packet!
   ackpacket((byte) 27);
   break;
...
```

Once this packet is acknowledged, the Garmin unit transmits the first of the available waypoints. The result is a `Pid_Wpt_Data` packet passed to the `processpacket` method. Waypoint formats vary from unit to unit, so this code simply dumps the raw bytes to the text area in numeric and character form:

```
...
case 35:
   trace.append("Handling waypoint data type.\r\n");
   // there are a many different varieties of waypoint
   // data structures, dependent on individual GPS unit and software version
   // this code just blasts any non-null data to the trace window.
   for (i=2;i<packet.length;i++)
   {
      if (packet[i] != 0x00) trace.append("\t" + i + "\t" + (byte)
              adjustbyte(packet[i]) + "\t" + (char) adjustbyte(packet[i]) + "\r\n");
   }
   ackpacket((byte)35);
   break;
...
```

When the final waypoint has been sent and acknowledged, the Garmin unit transmits a `Pid_Xfer_Cmplt` packet:

```
...
case 12:
   trace.append("Transfer Complete.\r\n");
   ackpacket((byte) 12);
   break;
...
```

The next panel presents the `acknowledgement` methods.

Ack and Nak

The Garmin protocol requires participants to issue acknowledgements and negative acknowledgements, as appropriate, in response to data received. The sample application implements these features in two methods, namely ackPacket and nakPacket. Here is the code listing for these functions:

```
public void ackPacket(byte packettype)
{
   byte[] request = new byte[8];
   trace.append("Sending ACK for packet type [" + packettype + "]\r\n");
   request[0] = 0x10;            // DLE
   request[1] = 0x06;            // Packet Id  (ACK)
   request[2] = 0x02;            // size of packet payload
   request[3] = (byte) packettype;  // type of packet we are acknowledging
   request[4] = 0x00;
   request[5] = (byte) (0x100 - (request[1] + request[2] + request[3] +
                        request[4]));
// checksum
   request[6] = 0x10;            // DLE
   request[7] = 0x03;            // ETX
   try
   {
      os.write(request);
      os.flush();
   }
   catch (Exception e)
   {
      trace.append("Error Sending ACK [" + e.getMessage() + "]\r\n");
      e.printStackTrace();
   }
   trace.append("ACK sent.\r\n");
}

public void nakPacket(byte packettype)
{
   byte[] request = new byte[8];
   trace.append("Sending NAK for packet type [" + packettype + "]\r\n");
   request[0] = 0x10;            // DLE
   request[1] = 0x15;            // Packet Id  (NAK)
   request[2] = 0x02;            // size of packet payload
   request[3] = (byte) packettype;  // type of packet we are acknowledging
```

```
   request[4] = 0x00;
   request[5] = (byte) (0x100 - (request[1] + request[2] + request[3] +
                                 request[4]));
// checksum
   request[6] = 0x10;            // DLE
   request[7] = 0x03;            // ETX
   try
   {
      os.write(request);
      os.flush();
   }
   catch (Exception e)
   {
      trace.append("Error Sending ACK [" + e.getMessage() + "]\r\n");
      e.printStackTrace();
   }
   trace.append("ACK sent.\r\n");
}
```

There is nothing particularly notable between these packets and the packets presented earlier. It is clear that a design opportunity for the `ibmdwgps` class is in the consolidation of the "boiler-plate" packet-send code. This and other suggestions are presented in the tutorial summary.

Section 7. Summary

Tutorial summary

This tutorial introduced the important GPS elements of *waypoints*, *routes*, and *tracks* and aimed at communicating the raw Hows and Whys of implementing the Garmin interface. The fundamental byte handling characteristics of Java code were explored and addressed with regard to the specifics of serializing a C structure across a communications channel.

The intent of the design for the sample application, `ibmdwgps`, was to teach the Garmin Interface, along with presenting various GPS data elements. The data types chosen provided a good mixture of data type handling including bytes, integers and strings. There is common functionality repeated through out many of the methods in the sample application; these are ideal candidates for optimization and better object-oriented design enhancements.

The next tutorial in this series examines the opportunities and challenges facing Java developers when creating GPS applications for the Palm OS environment.

Tutorial resources

- Download the latest Java SDK from *http://java.sun.com/j2se*.
- Download the `javax.comm` package from *http://java.sun.com/products/javacomm/index.html*. This package is required for this tutorial's sample application.
- *http://www.palm-communications.com/ibmdw* contains all of the source code presented in this tutorial.
- The *http://www.spacedaily.com* Web site provides a variety of space and satellite information and news.
- *http://www.apollocom.com/VisualGPS/* offers a "Freeware" GPS application which leverages the NMEA data protocol.
- *http://www.oreillynet.com/wireless/* regularly presents wireless development content, including GPS topics.
- You can find the introductory, prerequisite tutorial at *Introduction to GPS, Part 1*.
- A series of tutorials on Building Palm OS Applications, which discuss using a PDA for serial communications applications, is available on the developerWorks Wireless zone at *http://www-106.ibm.com/developerworks/wireless/library/wi-edpick.html* .
- *http://home.t-online.de/homje/RolandPf/gps_e.html* contains an application for interacting with Garmin devices. Included in the site is a GPLed source collection implementing the Garmin Interface.
- The tutorial, *Build your own Java library*, discusses Java library design principles
- You can find Garmin's communications interface protocol document at *http://www.garmin.com/support/commProtocol.html*. Additionally this same site

provides information on the specific data structures used for storing waypoints.

- Find Thomas Finley's description of two's complement calculation at *http://www.duke.edu/~twf/cps104/twoscomp.html*
- Elliotte Rusty Harold's book, *Java I/O*, published by O'Reilly and Associates *http://www.ora.com*, is a valuable resource to Java developers needing information and "How-To" knowledge in the realm of input and output streams in Java technology.

Feedback

Please send us your feedback on this tutorial. We look forward to hearing from you!

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The open source Toot-O-Matic tool is an XSLT stylesheet and several XSLT extension functions that convert an XML file into a number of HTML pages, a zip file, JPEG heading graphics, and two PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML. (It also saves our production team a great deal of time and effort.)

You can get the source code for the Toot-O-Matic at www6.software.ibm.com/dl/devworks/dw-tootomatic-p. The tutorial Building tutorials with the Toot-O-Matic demonstrates how to use the Toot-O-Matic to create your own tutorials. developerWorks also hosts a forum devoted to the Toot-O-Matic; it's available at www-105.ibm.com/developerworks/xml_df.nsf/AllViewTemplate?OpenForm&RestrictToCategory=11. We'd love to know what you think about the tool.