

Author: Jonas Kämpe, jonas@kampe.com
22 January 2005

Don't hog the GPS - How to make your application work together with other GPS software

Handheld computers are increasingly being hooked up to GPS devices, and the number of GPS-enabled applications is steadily increasing. This leads to a situation where several applications might be competing for a single GPS connection on a device. This article describes an easy and fast way to write polite GPS applications that can run alongside less considerate popular navigational software.

GPS applications have become big business and there are great opportunities for developers that find creative uses for GPS devices. Earlier proprietary systems are being challenged by cheaper and more flexible standards-based solutions.

Amateur pilots routinely want to combine access to their flight plans with local weather data, car drivers might want to switch between the built-in route planning and speed camera position services, and navigating the seas can be made safer combining a digital nautical map with travel logs and location-based forecasts on weather and winds.

So how do you do, if you don't want your users to have to close down one program to allow another to access a single GPS device? Sharing your GPS with other applications can be quite a challenge, and far from all applications are written with that scenario in mind.

Creating a virtual gate to the GPS

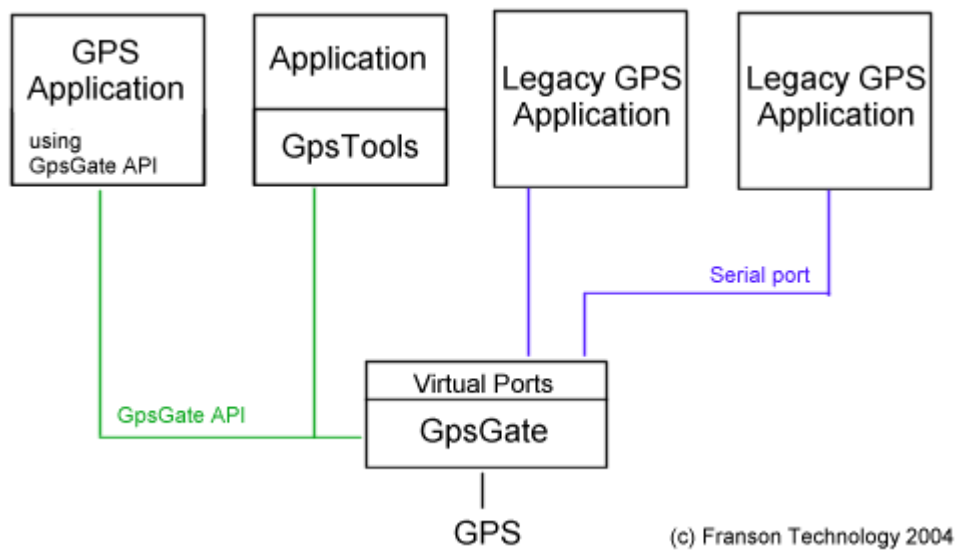
One well-known suite of GPS tools for developers comes from Franson Technology (<http://franson.com>). In particular there is a tool called Franson GpsGate, allowing end-users to run multiple GPS applications at the same time. With the software installed on a device, you can manually set up sharing of a GPS, and run TomTom Navigator alongside Mapopolis and other popular GPS applications.

This kind of sharing can also be accomplished programmatically by using something called the GpsGate SDK. It exposes an API to directly access the functionality of GpsGate from within your application, adding an intelligent layer between the application and the GPS. In that way, you won't be forced to write code to connect to the GPS device yourself, and it will save you from having to be aware of the exact settings of the GPS or the COM ports, all that information is handled by the GpsGate object. It also relieves your users from the hassle of having to manually configure the setup of the GPS connection.

By using the GpsGate API you make sure that your GPS application can co-exist with other applications from scratch. When your application is using the GpsGate API, all other GPS applications running on the device can still continue using the regular COM ports, or be set up to use the virtual COM ports provided by the GpsGate API.

Generally, only one application at a time can read a particular COM port for serial data. Handheld devices of today might have just one or two available COM ports for external devices, a fact that makes it even more important to share ports. The way GpsGate takes care of this shortcoming, is to create what is called “virtual COM ports”. It is basically a program that sits on top of a real COM port, making it appear to any interested applications as one or more serial ports. GpsGate allows for several applications to use and share one single virtual COM port, as well as splitting a COM port into several virtual ports.

If you are not into low-level serial port programming, you’ll be glad to know there are just a few lines of code involved in accessing and taking advantage of the GpsGate API.



Dot Net programming example: Sharing a GPS by using GpsGate

This example shows briefly how to open access to the GPS via GpsGate in a Dot Net Form and read position data from the device, as well as using GpsGate to act as a data source to other applications. This code example is for Windows Mobile devices, but GpsGate works well on any PC running Windows.

The easiest way to start developing using the GpsGate API is to download the GpsGate SDK from Franson Technology (http://franson.com/gpsgate/dev_guide.asp?platform=ppc). There you will find code examples for both Windows and Windows Mobile, in C# .Net, VB/eVB, and VC++/cVC++.

First, what you need to do in order to make the GpsGate API accessible to your .Net Compact Framework application is to reference the `GateApiNET.dll` found in the GpsGate SDK. All functions in the Dot Net GpsGate API are then available to programmers through an on object called `Simple`.

From within a form, all you have to do is to set up a variable for the `GateApi` object and create a new `Simple` object:

```
private GateApiNET.Simple m_simple = null;
```

```
m_simple = new GateApiNET.Simple();
```

Whenever GpsGate receives data from the GPS, it will generate an OnRead event, and we set our form to handle it:

```
m_simple.OnRead += new GateApiNET.OnRead(Form1_OnRead);
```

...and the button to open the GPS:

```
this.bOpen.Text = "Open the GPS";  
this.bOpen.Click += new System.EventHandler(this.bOpen_Click);  
this.Controls.Add(this.bOpen);
```

...and a label for displaying the read position data:

```
this.txtRead.Text = "";  
this.Controls.Add(this.txtRead);
```

The event handler for the “open”-button uses StartGpsGate() to make sure GpsGate is up and running, and specifies that we want to read the *output* channel of GpsGate. There is no need to specify a particular COM port or to have any other knowledge of the GPS:

```
private void bOpen_Click(object sender, System.EventArgs e)  
{  
    try  
    {  
        m_simple.StartGpsGate();  
        m_simple.Open("_output");  
    }  
    catch(Exception ex)  
    {  
        MessageBox.Show(ex.Message);  
    }  
}
```

Finally, we add the event handler that will be called each time new data is available from the GPS:

```
private void Form1_OnRead(string Data)  
{  
    if(Data == null)  
    {  
        // Connection is down. GpsGate was probably closed  
        MessageBox.Show("Connection down");  
        m_simple.Close();  
    }  
    else  
    {  
        txtRead.Text = Data;  
    }  
}
```

Your device should by now be connected to the application, and NMEA strings will appear in the text box as soon as the GPS gets a fix.

Adding more functionality: acting as a position source

GpsGate also provides a feature to act as a source of GPS data to other applications – basically your application can simulate an GPS.

In that case we would change the channel above into “_input” and add an event for writing, to send back an NMEA sentence into GpsGate and to all listening applications:

```
private void bWrite_Click(object sender, System.EventArgs e)
{
    try
    {
        m_simple.Write("$GPRMC,093339,A,5917.40531,N,01806.38798,
E,10.0,147.2,040607,0.0,E*4A\r\n");
    }
    catch(Exception ex)
    {
        MessageBox.Show(ex.Message);
    }
}
```

Higher-level GPS tools

In addition to offering a clean programmable application interface for accessing and sharing a GPS, users can benefit from the same features by setting up GpsGate manually. Something very handy during both debugging and demonstrations is the ability to log and retransmit NMEA sentences to other applications. You can also define a set of waypoints and let GpsGate simulate travel between those.

For readers that have not already developed a GPS application ready to be shared, Franson Technology also provides several other tools for rapid GPS application development. Most notably Franson GpsTools is a program that includes support for reading GPS data, converting between different coordinate systems and plotting to raster maps. It also integrates directly with GpsGate, making it a low-threshold introduction to GPS programming, with plenty of easy-to-follow examples in VB, .Net and C++.

By using tools such as these, you should be up and running with your GPS application development within a couple of hours, while still allowing for your users to mix and match between their favourite GPS applications!