Legal Notice

Copyright © 2004 Nokia Corporation. All rights reserved.

Nokia and Nokia Connecting People are registered trademarks of Nokia Corporation. Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. Other product and company names mentioned herein may be trademarks or trade names of their respective owners.

Disclaimer

The information in this document is provided “as is,” with no warranties whatsoever, including any warranty of merchantability, fitness for any particular purpose, or any warranty otherwise arising out of any proposal, specification, or sample. Furthermore, information provided in this document is preliminary, and may be changed substantially prior to final release. This document is provided for informational purposes only.

Nokia Corporation disclaims all liability, including liability for infringement of any proprietary rights, relating to implementation of information presented in this document. Nokia Corporation does not warrant or represent that such use will not infringe such rights.

Nokia Corporation retains the right to make changes to this specification at any time, without notice.

License

A license is hereby granted to download and print a copy of this specification for personal use only. No other license to any other intellectual property rights is granted herein.
Contents

1. Document Control ........................................................................................................... 5
   1.1 References .................................................................................................................. 5
   1.2 Documentation Conventions ..................................................................................... 5
   1.3 Abbreviations ............................................................................................................. 5
   1.4 Definitions ................................................................................................................. 5

2. Purpose ........................................................................................................................... 6
   2.1 List of the required interfaces .................................................................................... 6
   2.2 Constraints ................................................................................................................ 6

3. Technical Specification .................................................................................................... 7
   3.1 Type of the interface .................................................................................................. 7
   3.2 Interface class structure ............................................................................................ 7
      3.2.1 Client-server communication classes ............................................................... 7
      3.2.2 General position data classes ........................................................................... 8
      3.2.3 Extended data retrieval classes ........................................................................ 9
      3.2.4 Module information, status and status event classes ..................................... 10
      3.2.5 Privacy classes ................................................................................................ 11
      3.2.6 Update option classes ...................................................................................... 11
   3.3 Usage ......................................................................................................................... 12
      3.3.1 Protocol .............................................................................................................. 12
      3.3.2 Error handling .................................................................................................. 17
      3.3.3 Memory overhead ............................................................................................. 17
      3.3.4 Extensions to the Location Acquisition API .................................................... 17
   3.4 Example ...................................................................................................................... 17
      3.4.1 Simple location request ..................................................................................... 17
      3.4.2 Using HPositionGenericInfo .......................................................................... 18
   3.5 Detailed Description .................................................................................................. 19
      3.5.1 Standard client-server error codes ..................................................................... 19
      3.5.2 RPositionServer ............................................................................................... 20
      3.5.3 RPositioner ....................................................................................................... 24
      3.5.4 TCoordinate .................................................................................................... 32
      3.5.5 TLocality ......................................................................................................... 36
      3.5.6 TPosition ......................................................................................................... 39
      3.5.7 TPositionInfo .................................................................................................. 40
      3.5.8 HPositionGenericInfo ..................................................................................... 41
      3.5.9 TCourse ........................................................................................................... 49
      3.5.10 TPositionCourseInfo ..................................................................................... 51
      3.5.11 TSatelliteData ................................................................................................. 51
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.5.12</td>
<td>TPositionSatelliteInfo</td>
<td>52</td>
</tr>
<tr>
<td>3.5.13</td>
<td>TPositionModuleInfo</td>
<td>54</td>
</tr>
<tr>
<td>3.5.14</td>
<td>TPositionQuality</td>
<td>60</td>
</tr>
<tr>
<td>3.5.15</td>
<td>TPositionModuleStatus</td>
<td>63</td>
</tr>
<tr>
<td>3.5.16</td>
<td>TPositionModuleStatusEvent</td>
<td>66</td>
</tr>
<tr>
<td>3.5.17</td>
<td>TPositionUpdateOptions</td>
<td>70</td>
</tr>
<tr>
<td>3.5.18</td>
<td>CRequestor</td>
<td>73</td>
</tr>
<tr>
<td>3.5.19</td>
<td>RRequestorStack</td>
<td>76</td>
</tr>
<tr>
<td>3.6</td>
<td>Code architecture</td>
<td>77</td>
</tr>
</tbody>
</table>
1. **Document Control**

1.1 References


1.2 Documentation Conventions

Code is written with the **Courier New** font.

1.3 Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>A-GPS</td>
<td>Assisted GPS</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>DoP</td>
<td>Dilution of Position</td>
</tr>
<tr>
<td>E-OTD</td>
<td>Enhanced Observed Time Difference</td>
</tr>
<tr>
<td>GPS</td>
<td>Global Positioning System</td>
</tr>
<tr>
<td>MLFW</td>
<td>Mobile Location Framework</td>
</tr>
<tr>
<td>PSY</td>
<td>Positioning Technology Plug-in</td>
</tr>
<tr>
<td>QoP</td>
<td>Quality of Position</td>
</tr>
<tr>
<td>QoP</td>
<td>Quality of Position</td>
</tr>
<tr>
<td>WGS-84</td>
<td>World Geodetic System 1984</td>
</tr>
</tbody>
</table>

1.4 Definitions

<table>
<thead>
<tr>
<th>Mobile Location Framework</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A framework for enabling location based services in Symbian OS.</td>
</tr>
<tr>
<td>NMEA</td>
<td>National Marine Electronics Association. Defines the protocol for communication between marine instrumentation.</td>
</tr>
</tbody>
</table>
2. **Purpose**

The purpose of the Location Acquisition API is to enable terminal applications to retrieve information related to the current location of the mobile device. This API is included in the Series 60 Platform 2.6 and onwards.

In addition, the Location Acquisition API is capable of utilizing different types of hardware and positioning technologies to actually retrieve the location information.

The client application uses the Location Acquisition API to obtain estimates of the location the device is in at the moment. To get the information, the Location Acquisition API uses Location Server, which is only required to support the communication protocol used by the Location Acquisition API. Location Server performs all the operations to execute the client's requests. The Location Acquisition API makes the client application independent of client-server communication details.

The Location Acquisition API may be used by different types of applications utilizing location information and providing it in different forms to the user.

The Location Acquisition API defines a standard set of location information (latitude, longitude, altitude, accuracy and time) that most types of positioning technologies will be able to produce. However, the Location Acquisition API is also able to return extended information (such as speed) and also technology specific data (e.g. satellite data).

2.1 **List of the required interfaces**

The Location Acquisition API uses only those external interfaces and data types that have been defined in the Symbian OS. It uses only Mobile Location FW Location Server to provide location services to the client applications. However, even if Location Server does not exist on the system, certain parts of the Location Acquisition API (such as the position data classes) can still be used.

2.2 **Constraints**

The Location Acquisition API is valid for all platforms running on Symbian OS 7.0s or later.
3. Technical Specification

3.1 Type of the interface

The type of this interface is client-server interface.

The basis of the Location Acquisition API is standard Symbian OS client-server implementation. The client application first connects to Location Server and then opens a sub-session through which the location information is obtained. It provides both asynchronous and synchronous methods, which should be taken into account to avoid unexpected blocking of the client application.

3.2 Interface class structure

The Location Acquisition API contains a number of classes, which implement all the functionality of the interface. In the following sections, the classes are grouped according to their roles or according to functionality they provide together.

3.2.1 Client-server communication classes

Figure 3.1 presents the class hierarchy and methods in both RPositionServer and RPositioner. These classes implement the client-server communication with Location Server. All requests from the client are sent through these classes. The figure also contains the main classes and methods for changing the behavior of position updates.

![Diagram of client-server communication classes](image)

**Figure 3.1: Client-server communication classes**
RPositionServer represents a session to Location Server. It is intended for establishing a connection with Location Server and for obtaining general positioning module information. It is the primary class for all client applications wishing to access Location Server. It can also be used to obtain details of which positioning technologies are available and of their current status.

RPositioner is used for requesting location information, controlling update behavior and managing privacy. The purpose of RPositioner is to create a second level connection (i.e. a sub-session) through which location information is passed from Location Server.

### 3.2.2 General position data classes

Figure 3.2 shows the class hierarchy, the member data and significant methods for the standard position and data retrieval classes. Some members are not presented in the figure for the sake of clarity.

**Figure 3.2: General position data classes**

TPositionInfo is the main position data holder. It contains TPosition, which encapsulates TLocality and TCoordinate information. This is the standard...
parameter type passed to RPositioner when a client application requests location information. It is a simple wrapper class used to hold the identifier of the positioning technology that produced the fix and a TPosition class that holds the location information itself.

HPositionGenericInfo allows storing of arbitrary data and it can also be used for client-server data transfer.

TPositionInfoBase is the base class for all position information classes, both standard and extended. It is the operand of location request calls.

The TPositionClassTypeBase class is used for determining the data class on Location Server.

### 3.2.3 Extended data retrieval classes

Figure 3.3 describes the hierarchy of the extended information retrieval classes provided by the Location Acquisition API.

![Diagram](image)

**Figure 3.3: Extended data retrieval classes**

Extended data retrieval classes are used when specific location information is needed. The extended data retrieval classes are not required to be supported by every positioning module. The Location Acquisition API user can also define new extended data retrieval classes.

The TPositionCourseInfo class is used to get course information from a positioning module (if the module supports course information). It contains a TCourse class object, which represents the current speed of the device’s movement and the heading (direction) of the movement. It also provides information about the accuracy of these parameters.

TPositionSatelliteInfo gives physical information about the GPS network services used by the positioning device. This is a list of the visible and used GPS satellites.
satellites, their properties and common satellite information. The satellites are described by TSatelliteData.

A client does not usually modify these classes. Instead, they are filled in by Location Server or the positioning module.

### 3.2.4 Module information, status and status event classes

Figure 3.4 presents the classes for holding information about a positioning module and the significant methods for manipulating the data. The diagram also shows the main classes and methods for obtaining module status information.

Figure 3.4: Module information, status and status event classes

Module information may be important for the client because it influences the whole location retrieving process.

The TPositionModuleInfo class represents general information about a particular positioning module used by Location Server. In addition, the class contains a TPositionQuality class to describe the expected parameters of the location estimate production. This information can be used by the client or Location Server when looking for an appropriate module for performing special tasks.

TPositionModuleStatus (and its parent class TPositionModuleStatusBase) gives access to the current state of a particular module. A client may register for notifications on module status changes. In this case, it will receive information in the form of TPositionModuleStatusEvent (from the base class TPositionModuleStatusEventBase).
The TPositionQuality class defines quality parameters, which are also called Quality of Position (QoP) and are used for presenting positioning technology performance characteristics. See Section 3.5.14 for more details about the TPositionQuality class.

### 3.2.5 Privacy classes

Figure 3.5 depicts the main classes used to specify a chain of requestors.

![Privacy classes diagram](image)

**Figure 3.5: Privacy classes**

The RRequestorStack class is used to transfer a list of the requestors participating in the current location requests to Location Server. Each item in the list is defined by a CPosRequestor definition of the requestor.

### 3.2.6 Update option classes

Figure 3.6 depicts the main classes used to control location request related options.

![Update option classes diagram](image)

**Figure 3.6: Update option classes**
TPositionUpdateOptions is used by clients to define the behavior of position updates sent to them from Location Server. It is used as a parameter of RPositioner::SetUpdateOptions() and RPositioner::GetUpdateOptions().

3.3 Usage

3.3.1 Protocol

A client application utilizing the Location Acquisition API is typically interested in obtaining location estimates. To do that properly and effectively, the user needs to perform certain additional actions and know the possible ways to control how data is retrieved, how it is formatted and what particular data is provided. The Location Acquisition API provides the client with some additional functionality and information, which the client may need in the process of location retrieving. All these aspects are described in following sections, which describe the different major actions taken by an average client of the Location Acquisition API.

3.3.1.1 Client-server communication

As the Location Acquisition API is a client-server interface, a client has to set up a connection to Location Server by calling RPositionServer::Connect() to get location estimates. Accordingly to the common Symbian client/server API scheme, the RPositioner class represents a sub-session to Location Server. The class provides a set of OpenL() methods to define additional parameters of the newly opened sub-session: whether to use the default module, a particular module or a module that conforms to the client’s criteria (see Section 3.5.3 for more details).

Most location activity is done through RPositioner. The actions on getting general Location Server information are covered by RPositionServer. See the following paragraphs on how to use these classes.

To close a sub-session, the RPositioner::Close() method is provided. After the use of Location Server, RPositionServer::Close() must be called to close the client/server connection. All the used resources are freed at that moment.

The classes RPositionServer and RPositioner encapsulate methods for accessing Location Server. A number of these client-server requests are asynchronous. Although control quickly returns to the application, the TRequestStatus parameter variable is used to signal when Location Server has finished processing the request.

In the same way as with several other Symbian OS interfaces, only one request can be outstanding on each asynchronous method. For example, it is not allowed for an application to call NotifyPositionUpdate() before all the previous requests on that method have been completed. If the application wishes to call the same method before all the outstanding requests complete normally, it must first attempt to cancel those requests. The application must wait until the cancellation has been confirmed and then re-issue the desired request.

When a client application wishes to close one of its connections to Location Server, there can be no outstanding requests on that particular connection. If a client application attempts to close a connection before outstanding requests have been cancelled or completed, it is panicked. If the application wishes to close a RPositioner sub-session, it must ensure that it has successfully cancelled requests to GetLastKnownPosition() and
NotifyPositionUpdate(). If the application wishes to close the main
RPositionServer session, it must have successfully cancelled the
NotifyModuleStatusEvent() request. See Sections 3.5.2.5 and 3.5.3.2 for
more details.

3.3.1.2 Privacy handling

Determining of someone’s location is a question of the person’s privacy and
security. The Symbian OS and Positioning Framework provide a number of
safeguards to help prevent the misuse of location information. Location
information is not given to a requestor unless the user of the device for which
location is requested accepts it. For example, if user A requests the location of
user B, acceptance of user B is required to send the location information to user
A. User B can accept or deny the particular request, based on the requestor’s
information, which is provided by Positioning Framework (such as Mobile
Location FW). Positioning Framework requires this information from the
requestor, i.e. a client using the Location Acquisition API.

There are two requestor types: service requestors and contact requestors. A
service requestor represents any terminal or network application or service, and
a contact requestor represents an individual person requesting the location.
CRequestor::RequestorType() defines the type of the requestor.

The requestor information must be sent to Location Server before requesting
location estimates (otherwise, all such requests fail). This may be done in two
different ways:

1. **Single Requestor**: May be defined by RPositioner::SetRequestorL(),
   its three-argument variant. In this case, the requestor is defined directly by
   the values supplied to the function’s parameters. This case is used by
   applications or services if they are the only request source.

2. **Stack of Requestors**: Specifying requestor information through the stack of
   requestors by calling the second variant of
   RPositioner::SetRequestorL(), which gets an argument of the type
   RRequestorStack. This approach is used more often when a request is
   made by an application or service on behalf of some other entity such as a
   contact or some other application.

RRequestorStack is a container for a list of the requestors on behalf of which
the request is made. Each particular requestor in the stack is represented by a
CRequestor class.

**Note**: Regardless of the number of requestors, there is always one service
requestor, because the application executing the location request must be
always shown, even if it made the request on behalf of a contact requestor. For
this reason, the last item in the requestor stack must always be a service
requestor. See detailed information on CRequestor in Section 3.5.18 to get
more information on the requestor’s data. Also in the Single Requestor case, the
requestor type must always be a service.

See Section 3.5.3 for more details on the SetRequestorL function.

3.3.1.3 Simple obtaining of a location estimate

In order to perform a simple location request, a client calls the asynchronous
RPositioner::NotifyPositionUpdate() function. As the argument of
the function, any class derived from TPositionInfoBase may be used. The normal
use is to provide a TPositionInfo class object containing the simplest location
information, or an HPositionGenericInfo class object, which is more flexible
in that it allows the transmission of arbitrary data between the client and Location Server.

The Location Acquisition API can cater for extended or device specific information available from different position technologies. The information within the HPositionGenericInfo class is accessed via numeric field identifiers using the templated GetValue() and SetValue() methods. Some of these field identifiers have already been reserved (e.g. speed and course) while others are available for suppliers of positioning technologies to define. A client application can request a variety of location related information via this class; it is the responsibility of the position technology to return as much of the requested data as possible.

All positioning modules must support these TPositionInfo and HPositionGenericInfo classes. However, it is also possible to use other classes for receiving additional information if the used module supports additional information. See Section 3.3.1.5 for details.

Upon the completion of the NotifyPositionUpdate function, the passed argument is filled with the data provided by a location module (plug-in). TPositionInfo basically contains a TPosition object, which in turn holds TLocality and also TCoordinate data. The most important properties are the following:

- TCoordinate: Altitude(), Latitude(), Longitude(), Datum();
- TLocality: HorizontalAccuracy(), VerticalAccuracy();
- TPosition: Time();

See the detailed description for the classes TPosition (see Section Error! Reference source not found.), TLocality (see Section Error! Reference source not found.) and TCoordinate (see Section Error! Reference source not found.).

TPositionInfoBase (the base class for all position-holding classes) contains information about the module that produced the estimate and the position update type (see Section 3.3.1.4).

If the client needs more estimates, it calls the RPositioner::NotifyPositionUpdate() function again. If the client is interested in sequential periodic updates, it can use the location update options (see Section 3.3.1.4) to define that.

Another way to get a position is the RPositioner::GetLastKnownPosition() function. This function eliminates the need to wait for a new estimate if there is already one in the cache. Commonly, TPositionInfo or HPositionGenericInfo are used as arguments as they are the only classes required to be supported by positioning modules.

Note that position update request and last known position request are mutually exclusive. This means that it is not allowed to have them both outstanding within same sub-session. See Sections 3.5.3.5 and 3.5.3.6

3.3.1.4 Location update options

In some cases, a client may need to specify more criteria for how location updates should be generated, what data should be considered as valid or not and so on. The RPositioner::SetUpdateOptions() function allows to specify such criteria via the TPositionUpdateOptions class. Location update
options affect only RPositioner::NotifyPositionUpdate(). The TPositionUpdateOptions class provides the following options:

- **Update Interval**: In case a client needs to receive periodic location updates (e.g. every 20 seconds, or every hour), it can specify the parameter and issue the first request. After the completion of the current request, the client immediately issues the next request and it is guaranteed to receive the next location update upon the elapse of the specified interval.

- **Update Timeout**: Guarantees a client that its location request will not last longer than the specified time interval. The request either completes successfully, or, if the positioning module is not able to produce the next estimate within the given interval, the location request is cancelled by the timeout.

- **Update Maximum Age**: Defines the maximum age of a cached estimate returned by a module. Specifying this value defines that the client can accept location information, which was produced some time before the request was issued. More exactly, when the client receives the estimate, it is guaranteed that the estimate was generated within the specified time period in the past, not earlier. The default value is zero, meaning that the client accepts only newly generated positions. The value must be less than Update Interval.

- **Partial Updates**: Informs the module that the client application is willing to accept incomplete location information. An example of where this might be useful is in GPS where information on the observed satellites is available before the devices’ location can be determined.

See Section **Error! Reference source not found.** for more details.

### 3.3.1.5 Extended location information

In addition to the general position classes introduced in Section 3.3.1.3, there are two other position classes: TPositionCourseInfo and TPositionSatelliteInfo.

TPositionCourseInfo contains a TCourse object, which may be used to get information such as the current speed at which the device is moving, the device’s direction in terms of bearing, and also the accuracy of these parameters.

TPositionSatelliteInfo contains information about the satellites used in locating: how many satellites are available, which of those were used to produce the current estimate and a list of the satellites with their data. A satellite is represented by the TSatelliteData class.

A client gets the information by sending the appropriate object to Location Server via NotifyPositionUpdate. If the currently used module supports the data (see Section 3.3.1.6), the module fills the objects with the data and returns it to the client. TPositionSatellites inherits TPositionCourseInfo, which is derived from the TPositionInfo class, and thus the client gets the extended information simultaneously with the general data in one request. However, if the module does not support the requested data class, the whole request fails. Therefore, the client should ensure if the particular position information class is supported by the current module before it issues a request. For more information, see Section 3.3.1.6.

### 3.3.1.6 Module information

RPositionServer also provides the methods GetModuleInfoByIndex() and GetModuleInfoById() to get information about the currently available
modules. Modules are identified by the TPositionModuleId class and its data is provided in the format of the TPositionModuleInfo class. To know the state of a particular module at a given moment, GetModuleStatus() may be called. See Section Error! Reference source not found. for information on the module status. To receive all module status events, the NotifyModuleStatusEvent() member function is used.

The module information is delivered in the form of the TPositionModuleInfo class. The most important fields are the following:

- **Position Quality**: Used for retrieving the module’s Quality of Position parameters, see Section 3.5.13.5.

- **Capabilities**: Defines which particular information may be obtained from the module, e.g. direction, speed, address, etc. Any information the module is capable of providing may be obtained via HPositionGenericInfo.

- **Classes Supported**: Includes important information as the client needs to know which classes are supported before using extended location information classes in requests to the module. See LbsClassTypes.H for a list of the position class types constants.

- **Module ID and Module Name**: May be used for further references.

The TPositionModuleEvent class is used to hold information about the module when a client is informed about events in the module. This is also used by clients to specify of which types of events happening in the module they wish to be notified. Clients use this class to determine what particular events have just occurred and the new status of the module.

### 3.3.1.7 Units of measurement and co-ordinates

The Location Acquisition API defines a number of standard classes for holding position information. Unless otherwise stated, these classes use the following measurement system:

- Latitude and longitude are held in 64 bit real numbers. Altitude is held in a 32 bit real number. By default, the WGS-84 datum is used to reference co-ordinates. All positions returned by Location Server use this co-ordinate reference system. Latitude values are in the range [-90, +90] (-90 <= Lat <= +90) whereas Longitude values are in the range [-180, +180] (-180 <= Long < +180)

- Both horizontal and vertical accuracy are held in 32 bit real numbers that represent the error offset in meters. In practice, the actual position lies within the respective accuracy circle with 68% probability.

- Bearings and headings are held in 32 bit real numbers and represent the direction in degrees from the true north. Their corresponding accuracies are also held in 32 bit real numbers and represented in degrees.

### 3.3.1.8 Unassigned values

A number of the classes described in this document use real numbers to hold position, accuracy and other information. Sometimes some values can be unset. Altitude is a typical example.

Where a real number has not been assigned, Mobile Location FW indicates this by a special (reserved) bit pattern. This bit pattern is defined by the Symbian OS.
and is usually referred to as “Not a Number” (\( NaN \)). Applications check if a value is \( NaN \) by using the standard method from the maths library (\( Math::IsNaN \)).

Although \( NaN \) can be tested, the value cannot be used in calculations. On target devices, attempting to use a \( NaN \) value in an arithmetic expression results in an application panic (USER-EXEC 3). On the emulator, the panic may not occur, but it is still illegal to use the value in calculations.

3.3.2 Error handling

The Location Acquisition API uses the standard Symbian error reporting mechanism. In case of an irrecoverable error, panics are used. Otherwise, leaves are used or functions return error codes as their return values. The Location Acquisition API uses standard system error codes plus certain errors defined in the Location Acquisition API. Those can be found in the \( LbsErrors.H \) file.

The Location Acquisition API methods and the related error handling is described in the Section 3.5. A client handles these errors in the same way as normally in Symbian platform applications.

3.3.3 Memory overhead

When using the Location Acquisition API, the memory overhead is dependent on the amount of instantiated classes but there are also some cases when extra memory usage can be involved. Nevertheless, the memory usage is always controlled by the client application and there are never any uncontrolled allocations of large amounts of memory that the client cannot prevent or avoid.

When communicating with Location Server, additional buffers are used to for transferring data (mostly related to the requestor information). Also the \( HPositionGenericInfo \) class uses a heap to store location data. However, in all cases, the amount of memory is defined by the client and thus can be exactly predicted. After use, the needed additional buffers (if any) are unallocated automatically.

The number of simultaneous sessions and sub-sessions also has an impact on the overall memory usage. For this reason, unnecessary sessions and sub-sessions should be avoided.

3.3.4 Extensions to the Location Acquisition API

The Location Acquisition API can be extended by defining new data classes. It is possible to provide new location estimate data structures by deriving new classes from \( TPositionInfoBase \) or any of its descendants. A positioning module supporting such a class provides the client with new information. For details, see Section Error! Reference source not found..

3.4 Example

3.4.1 Simple location request

The following example shows the code, which a client should execute to get a position update.

```c
#include <lbs.h>

// Init Connection
```
RPositionServer server;
RPositioner positioner;
User::LeaveIfError(server.Connect());
CleanupClosePushL(server);
User::LeaveIfError(positioner.Open(server)); // use default positioning module
CleanupClosePushL(positioner);

// Specify Requestors
_LIT(KCntPhone, "+358501234567");
_LIT(KSrvName, "MyService");
RRequestorStack stack;
CRequestor* contact = CRequestor::NewLC(CRequestor::ERequestorContact,
CRequestor::EFormatTelephone, KCntPhone);
stack.Append(contact);
CRequestor* service = CRequestor::NewLC(CRequestor::ERequestorService,
CRequestor::EFormatApplication, KSrvName);
stack.Append(service);
User::LeaveIfError(positioner.SetRequestor(stack));

// Issue a Location Request
TRequestStatus status;
TPositionInfo posInfo;
positioner.NotifyPositionUpdate(posInfo, status);
User::WaitForRequest(status);
User::LeaveIfError(status.Int());

// Analyze Results
TPosition position;
posInfo.GetPosition(position);

// Issue a new Location Request
positioner.NotifyPositionUpdate(posInfo, status);
User::WaitForRequest(status);
User::LeaveIfError(status.Int());

// Cleanup
stack.Reset();
CleanupStack::PopAndDestroy(service);
CleanupStack::PopAndDestroy(contact);
CleanupStack::PopAndDestroy(&positioner); // this will call Close() method
CleanupStack::PopAndDestroy(&server); // this will call Close() method

3.4.2 Using HPositionGenericInfo

The usage of HPositionGenericInfo in location requests is the same as the usage of TPositionInfo. The only exception is that the client has to specify what kind of information it expects from Location Server before issuing the request. This is specified by the setting identifiers of the requested fields in the HPositionGenericInfo object (see Section 3.3.1.3):

// Init Connection
RPositionServer server;
RPositioner positioner;
User::LeaveIfError(server.Connect());
CleanupClosePushL(server);

TPositionModuleId moduleId;
moduleId.iUid = KModuleIdThatSupportsAddressCapabilites; // select valid module

Version 6.0 | April 22, 2004
User::LeaveIfError(positioner.Open(server, moduleId)); // use specific module
cleanupClosePushL(positioner);

// Init Generic Info buffer
HPositionGenericInfo* genInfo = HPositionGenericInfo::NewLC();
// default sizes
genInfo->SetRequestedField(EPositionFieldCountry);
genInfo->SetRequestedField(EPositionFieldCity);

// Specify Requestors
_LIT(KCntPhone, "+358501234567");
_LIT(KSrvName, "MyService");
RRequestorStack stack;
CRequestor* contact = CRequestor::NewLC(
    CRequestor::ERequestorContact,
    CRequestor::EFormatTelephone,
    KCntPhone);
stack.Append(contact);
CRequestor* service = CRequestor::NewLC(
    CRequestor::ERequestorService,
    CRequestor::EFormatApplication,
    KSrvName);
stack.Append(service);
User::LeaveIfError(positioner.SetRequestor(stack));

// Make a request
TRequestStatus status;
positioner.NotifyPositionUpdate(*genInfo, status);
User::WaitForRequest(status);
User::LeaveIfError(status.Int());

// Get Results
TPtrC16 city, country;
if (genInfo->IsFieldAvailable(EPositionFieldCountry))
{
    genInfo->GetValue(EPositionFieldCountry, country);
}
if (genInfo->IsFieldAvailable(EPositionFieldCity))
{
    genInfo->GetValue(EPositionFieldCity, city);
}

// Cleanup
stack.Reset();
cleanupStack::PopAndDestroy(service);
cleanupStack::PopAndDestroy(contact);
cleanupStack::PopAndDestroy(genInfo);
cleanupStack::PopAndDestroy(&positioner); // this will call Close() method
cleanupStack::PopAndDestroy(&server); // this will call Close() method

3.5 Detailed Description

This section contains detailed information on the Location Acquisition API’s classes and their methods and usage. The methods may be arranged in groups accordingly to their roles.

3.5.1 Standard client-server error codes

Many of the methods contained in the Location Acquisition API involve communication with Location Server. Rather than duplicating the standard error return codes for each of these functions, a summary is presented below.
However, please note that other (more unusual) error codes are also possible and the application programmer should expect more than the standard error codes presented in this document.

- **KErrBadHandle**: Returned if the connection to Position Server is not present or is no longer valid.
- **KErrServerBusy**: This value is returned if Location Server has taken too long to accept a request.

### 3.5.2 RPositionServer

This is generally the first interface class used by all client applications. It is used to make the primary connection to Location Server. After the primary connection has been established, its handle is passed as a parameter of the Open methods of RPositioner to create a sub-session.

The RPositionServer class is also used to obtain details about the available positioning modules and to obtain notification of events regarding their status.

#### 3.5.2.1 Connecting to Location Server

**Methods:**

```cpp
TInt Connect()
```

**Usage:**

`Connect()` establishes the primary link (session) to Location Server. It must be called before an attempt to obtain module information or on opening a sub-session, e.g. `RPositioner::Open()`.

**Return values:**

`Connect()` returns `KErrNone` when a link has been successfully established with Location Server.

**Error codes:**

Standard client/server error codes, see Section *Error! Reference source not found.* for more details.

**Panics:**

A `EPositionServerHandleNotClosed` panic occurs if a connection is already present.

#### 3.5.2.2 Closing the connection

**Methods:**

```cpp
void Close()
```

**Usage:**

`Close()` terminates a connection with Location Server and reclaims any used memory.
Before the connection to Location Server is closed, the client application must ensure that any outstanding notification requests have been cancelled. In particular, applications must issue all appropriate cancel requests and then wait for a confirmation that the notifications have been terminated. A failure to do so results in a client side panic.

A client application should also ensure that all sub-sessions are closed before the connection to Location Server is closed.

Panics:

The panic EPositionRequestsNotCancelled occurs if the client application still has requests outstanding with Location Server.

3.5.2.3 Available positioning modules

Methods:

| TInt GetNumModules  |
| (TUint& aNumModules) const |
| TInt GetModuleInfoByIndex |
| (TInt aModuleIndex, TPositionModuleInfoBase& aModuleInfo) const |
| TInt GetModuleInfoById |
| (TPositionModuleId aModuleId, TPositionModuleInfoBase& aModuleInfo) const |
| TInt GetDefaultModuleId |
| (TPositionModuleId& aModuleId) const |

Usage:

GetNumModules() is used to obtain the current number of available positioning modules. On a successful completion, the parameter aNumModules returns the count.

GetModuleInfoByIndex() takes an index in the range 0 to \{(GetNumModules()-1)\} as its first parameter and returns details about that module in aModuleInfo.

GetModuleInfoById() takes a module identifier as its first parameter and returns details about that module in aModuleInfo.

GetDefaultModuleId() returns the identifier of the system's default positioning module. This module will be used if no criteria is specified when an RPositioner sub-session is created. See Section 3.5.3.1 for more details.

Return values:

All methods return KErrNone if the information has been successfully retrieved from Location Server.

Error codes:

These methods return KErrNotFound if the specified module parameter is incorrect. This may be due to the specified module no longer being available. For example, it may have been uninstalled.

Standard client-server error codes, see Section Error! Reference source not found. for more details.

Panics:
These routines panic with EPositionServerBadHandle if no connection has been established with Location Server.

### 3.5.2.4 Module status

**Methods:**

```cpp
TInt GetModuleStatus(TPositionModuleStatusBase& aPosModuleStatus, TPositionModuleId aModuleId) const

void NotifyModuleStatusEvent(TPositionModuleStatusEventBase& aStatusEvent, TRequestStatus& aStatus, TPositionModuleId aModuleId = KPositionModuleIdAny) const
```

**Usage:**

GetModuleStatus() returns information about the specified positioning module. See Section Error! Reference source not found. for more details.

NotifyModuleStatusEvent() is an asynchronous method, which reports status changes of either a single module or all positioning modules. If the parameter aModuleId is not specified, then status updates are provided for all positioning modules. If aModuleId is supplied, then the application only receives status updates for the specified module.

After a notification has been received, the client application must re-issue the status change request if it wishes to obtain further updates.

To cancel an outstanding NotifyModuleStatusEvent() request, the method RPositionServer::CancelRequest is called with the parameter aRequestId set to the value EPositionServerNotifyModuleStatusEvent.

Before a client application closes the connection to Location Server, it must first ensure that all outstanding request (such as NotifyModuleStatusEvent()) have been cancelled. In particular, after the application has invoked the CancelRequest() method, it must wait until Location Server has reported that the request has indeed been cancelled.

**Return values:**

GetModuleStatus() returns KErrNone if the current module status has been successfully retrieved.

In the case of NotifyModuleStatusEvent(), aStatus returns the result code. KErrNone indicates that a status update has been received.

**Error codes:**

GetModuleStatus() returns KErrNotFound if the specified module does not exist.

In the case of NotifyModuleStatusEvent, aStatus returns one of the following result codes:

KErrNotFound if the specified module does not exist.

KErrArgument if the requested event mask is zero (i.e. no events have been requested).

KErrCancel if the request has been successfully cancelled.
Panics:

NotifyModuleStatusEvent panics with EPositionServerBadHandle if no connection has been established with Location Server.

NotifyModuleStatusEvent panics with EPositionDuplicateRequest if the client already has a module status request outstanding on the same session with Location Server.

3.5.2.5 Canceling requests

Methods:

```cpp
TInt CancelRequest (TInt aRequestId)
```

Usage:

This method attempts to cancel an outstanding asynchronous request from within the RPositionServer class. Canceling requests is typically attempted when an application is closing down.

The aRequestId parameter specifies the request to be cancelled. For example, to cancel a call to NotifyModuleStatusEvent(), the parameter must be set to EPositionServerNotifyModuleStatusEvent.

As CancelRequest() is an asynchronous method, Location Server typically queues the cancellation. Generally, Location Server attempts to abort the specified request shortly after the Cancel method has returned.

Return values:

KErrNone is returned if the Cancel request has been successfully sent to Location Server. However, it does not mean that the specified request has actually been terminated. This is indicated via the request status of the specified request.

Error codes:

KErrNotFound is returned if there is no outstanding request with the specified ID.

Standard client/server error codes, see Section Error! Reference source not found. for more details.

Panics:

A EPositionServerBadHandle occurs if no connection has been established with Location Server.

3.5.2.6 Version details

Methods:

```cpp
TVersion Version() const
```

Usage:

Version() is used to obtain the current version number of the Location Acquisition API.
Return values:
The method returns an instance of the standard TVersion class.

3.5.3   RPositioner

RPositioner is used to create a sub-session with Location Server for the
purpose of obtaining the current position. In addition to actually obtaining the
position information, this class also provides mechanisms for obtaining the last
known position and for changing how often the client wishes to receive position
updates as well as for identifying itself to Mobile Location FW.

Before using the class, a primary connection must have already been
established with Location Server (see Section Error! Reference source not
found. for more details).

3.5.3.1 Opening a sub-session

Methods:

| TInt Open(RPositionServer& aPosServer)   |
| TInt Open(RPositionServer& aPosServer, const TPositionCriteriaBase& aCriteria) |
| TInt Open(RPositionServer& aPosServer, TPositionModuleId aModuleId) |

Usage:
The Open() methods are used to open a sub-session with Location Server in
order to retrieve position information.

After successfully establishing an RPositioner sub-session, the client
application must call RPositioner::SetRequestor to identify itself and (if
appropriate) on who’s behalf it is retrieving the location information. See
Sections 3.3.1.2 and 3.5.3.3 for more details.

All Open() methods take as their first parameter (aPosServer) a handle to an
RPositionServer object, which must have a successfully established
connection with Location Server before the Open() method can be used. See
Section 3.5.3 for more details.

Open(RPositionServer& aPosServer) connects the client application to the
default positioning module. The ID of this module can be obtained using
RPositionServer::GetDefaultModuleId().

Open(RPositionServer& aPosServer, TPositionModuleId aModuleId) allows the client application to specify that it wishes to receive location
information from a specified module (aModuleId). The identifier for the module
is generally determined by obtaining the list of the available modules from
RPositionServer.

The Open(RPositionServer& aPosServer, const
TPositionCriteriaBase& aCriteria) method is not supported in Series 60
Platform release 2.6. Also the related criteria classes such as
TPositionCriteriaBase are not documented in this specification.

Return values:
All Open() methods return KErrNone when an RPositioner sub-session has
been successfully established with Location Server.

Error codes:
KErrNotFound is returned if the module identifier (aModuleId) is invalid. A previously correct module identifier may become invalid if the positioning module has been uninstalled or brought off-line, e.g. by a user intervention via the control panel.

KErrNotFound is also returned if no positioning module satisfies the required criteria (aCriteria).

KErrNotFound is also returned in Series 60 Platform release 2.X when the Open(RPositionServer& aPosServer, const TPositionCriteriaBase& aCriteria) method is called.

Also standard client-server error codes can be returned. See Section 3.5.1 for more details.

**Panics:**

The panic EPositionServerHandleNotClosed occurs if a connection has already been established with Location Server.

### 3.5.3.2 Closing a sub-session

**Method:**

```cpp
void Close()
```

**Usage:**

Close() must be called when the RPositioner sub-session is no longer required. This function performs memory and connection cleanup.

Before a sub-session is closed, the client application must ensure that all outstanding notification requests have been cancelled. In particular, the application must issue all the appropriate Cancel requests and then wait for a confirmation that the notifications have been terminated. A failure to do so results in a panic.

**Panics:**

The panic EPositionRequestsNotCancelled occurs if the client application still has requests outstanding with Location Server.

### 3.5.3.3 Specifying requestors

**Methods:**

```cpp
TInt SetRequestor
(const RRequestorStack& aRequestorStack)
TInt SetRequestor
(CRequestor::TRequestorType aType,
 CRequestor::TRequestorFormat aFormat,
 const TDesC& aData)
```

**Defined:**

RPositioner (this class).

**Usage:**
Both overloads of `SetRequestor()` are used by applications to supply the identity of who (or what) is requesting location information. Typically, this is the application itself, but in some cases it may also be a remote party.

One of these methods must be called after a positioning sub-session is created with `RPositioner::Open()` and before any requests to obtain location information.

These operations do not invoke a privacy check at this stage. A privacy check is carried out when the application actually requests location information using either the `GetLastKnownLocation()` or `NotifyPositionUpdate()` methods. These methods return the status of `KErrAccessDenied` if the privacy check determines that any of the specified requestors are not allowed to retrieve the devices’ location.

Standard types of applications use the first version of `SetRequestor()` and call it only once during a sub-session. A typical call takes the following form, where `positioner` is an object of the type `RPositioner`:

```
_LIT(KAppName, "My App Name");
positioner.SetRequestor(CRequestor::ERequestorService, CRequestor::EFormatApplication, KAppName);
```

If the client application is retrieving location information on behalf of either another (local) application or a remote party, it must use the second version of `SetRequestor()` to identify both itself and the other parties. Supplying the identity of a number of parties is done via the `RRequestorStack` class. If the other party changes for some reason, then it is the responsibility of the client application to update the requestor information with another call to `RPositioner::SetRequestor()`. The application itself must be the last item in the requestor stack. The format `EFormatApplication` may not be used with the `ERequestorContact` requestor type.

See Sections 3.5.19 and 3.5.18 for details about the classes `RRequestorStack` and `CRequestor` and Section Error! Reference source not found. for general information on privacy handling.

**Return values:**

`KErrNone` is returned on a successful completion of the call.

**Error codes:**

`KErrArgument` is returned if the format or type of the specified requestor parameter (`aRequestor`) is incorrect, or if the `EFormatApplication` format is used with an `ERequestorContact` type of requestor.

The overload `SetRequestor(const RRequestorStack& aRequestorStack)` returns standard error codes for conditions such as out of memory (e.g. `KErrNoMemory`).

`KErrArgument` is returned if requestor data (`aData` argument) length exceeds 255 characters for any of specified requestors.

**PANICS:**

The panic `EPositionServerBadHandle` occurs if no connection has been established with Location Server.
3.5.3.4 Update options

Methods:

```cpp
TInt GetUpdateOptions(TPositionUpdateOptionsBase& aPosOption) const
TInt SetUpdateOptions(const TPositionUpdateOptionsBase& aPosOption)
```

Usage:

SetUpdateOptions() is used by the client application to modify the behavior of RPositioner::NotifyPositionUpdate() requests. For example, it enables the client to request an interval time for receiving position updates. See Section Error! Reference source not found. for more details.

If the client application has an outstanding NotifyPositionUpdate() request when SetUpdateOptions() is called, the behavior of the notification is not affected by the new set of update options. The new options only apply to the subsequent notification requests.

GetUpdateOptions() retrieves the current update options being used by an RPositioner sub-session.

Return values:

Both methods return KErrNone on successful completion.

Error codes:

SetUpdateOptions() returns KErrArgument if the specified options conflict. For example, if the timeout period is less than the specified update interval.

SetUpdateOptions() returns KErrNotSupported if the positioning module is unable to support the required options. For example, if the update interval period is too short.
Panics:

The panic EPositionServerBadHandle occurs if no connection has been established with Location Server.

3.5.3.5 Last known position

Methods:

```c
void GetLastKnownPosition(TPositionInfoBase& aPosInfo, TRequestStatus& aStatus) const
```

Usage:

This method returns the cached position information if it is available. This method can be an efficient mechanism in terms of speed, cost and power consumption for obtaining the devices’ recent position.

If the device’s position has not previously been discovered, this method does not instigate a new position fix. It is therefore possible that this method fails to return a fix.

If the method successfully retrieves the last known position, the client application should always check its associated timestamp to ensure that the last known position is still relevant. It may also wish to inspect the accuracy of the location fix.

As with all attempts to retrieve the device’s position, Mobile Location FW carries out a privacy check to determine if the requestor is allowed to carry out this operation. The client application must have previously called RPositioner::SetRequestor() to specify who (or what) is requesting the information.

The method is asynchronous: on completion, aStatus contains the return status.

On successful completion, aPosInfo contains the position information (see Section Error! Reference source not found. for more details).

**Note:** The parameter aPosInfo must be of the type TPositionInfo as this is the only class that positioning modules and Mobile Location FW are required to support.

The client application may attempt to cancel this asynchronous request by calling RPositioner::CancelRequest(EPositionerGetLastKnownPosition).

Before a client application closes a sub-session, it must first ensure that all outstanding requests (such as GetLastKnownPosition()) have been cancelled. In particular, after the application has invoked the CancelRequest() method, it must wait until Location Server has reported that the request has indeed been cancelled.

Return values:

aStatus returns KErrNone on successful completion.

Error codes:

aStatus returns KErrUnknown if no cached position information is available.
aStatus returns KErrArgument if the parameter aPosInfo is of a non-supported type. The only parameter type that is guaranteed to be supported is TPositionInfo.

aStatus returns KErrAccessDenied if no requestor information has been specified by the client application. See Section 3.5.3.3 for more details.

aStatus also returns KErrAccessDenied if the privacy check determines that any of the specified requestors do not have permission to retrieve location information. See Section 3.5.3.3 for more details.

aStatus also returns KErrPositionBufferOverflow if there is insufficient space to return the required information back to the client. This situation can occur when using HPpositionGenericInfo if the application has not allocated a large enough buffer.

aStatus returns KErrTimedOut if the user was required to authorize the request, but the user did not respond within the timeout period. The timeout period is set using RPositioner::SetUpdateOptions(). See also Section Error! Reference source not found. for more details.

aStatus returns KErrCancel if the request has been successfully cancelled.

Panics:
The panic EPositionServerBadHandle occurs if no connection has been established with Location Server.

The panic EPositionDuplicateRequest occurs if the client already has an outstanding request to get the last known position or to obtain position updates (RPositioner::NotifyPositionUpdate()) on the same RPositioner sub-session.

3.5.3.6 Obtaining the current position

Methods:

```cpp
void NotifyPositionUpdate(TPositionInfoBase& aPosInfo, TRequestStatus& aStatus) const
```

Usage:

This is an asynchronous method for obtaining position updates. On successful completion, aPosInfo holds information on the device’s current position. It is possible to pass any class that is derived from TPositionInfoBase. The standard data retrieval class is TPositionInfo, which all positioning modules must support.

Extended information can be obtained by using one of the specialized data retrieval classes, for example TCourseInfo (see Section Error! Reference source not found.).

If the required information is not contained in one the extended data retrieval class, the application can use the generic data retrieval class HPpositionGenericInfo. This class defines a larger set of information fields than the extended data retrieval classes. However, the application must again stipulate its data requirements when the sub-system is created (via the RPositioner::Open() method).

After a position update notification has been received, the client application must re-issue the request if it wishes to obtain further updates.
As with all attempts to retrieve the device’s position, Mobile Location FW carries out a privacy check to determine if the requestor is allowed to carry out this operation. The client application must have previously called RPositioner::SetRequestor() to specify who (or what) is requesting the information.

The client application may attempt to cancel this asynchronous request by calling RPositioner::CancelRequest(EPositionerNotifyPositionUpdate).

Before a client application closes a sub-session, it must first ensure that all outstanding requests (such as NotifyPositionUpdate()) have been cancelled. In particular, after the application has invoked the CancelRequest() method, it must wait until Location Server has reported that the request has indeed been cancelled.

Return values:

aStatus returns KErrNone on successful completion.

aStatus returns KPositionQualityLoss if the returned position information does not meet the quality criteria specified by the application when the sub-session was created via the RPositioner::Open() method. This is generally a transient problem due to a temporary loss of accuracy.

aStatus returns KPositionPartialUpdate if position information has been retrieved but it is incomplete. **Note:** KPositionPartialUpdate is a positive value and does not represent an error. This situation only occurs if the application has explicitly chosen to accept partial updates. See Section Error! Reference source not found. for more information.

Error codes:

aStatus returns KErrTimedOut if the requested location information could not be retrieved within the maximum period as specified in the current update options. This may occur if there are delays in the positioning device, or if the user was required to authorize the request, but the user did not respond within the timeout period. See Sections 3.5.3.4 and Error! Reference source not found. for more details.

aStatus returns KErrArgument if the positioning module is unable to support the type of the class passed in aPosInfo. All positioning modules are required to support both TPositionInfo and HPositionGenericInfo.

aStatus returns KErrAccessDenied if no requestor information has been specified by the client application. See Section 3.5.3.3 for more details.

aStatus also returns KErrAccessDenied if the privacy check determines that any of the specified requestors do not have permission to retrieve location information. See Section 3.5.3.3 for more details.

aStatus also returns KErrPositionBufferOverflow if there is insufficient space to return the required information back to the client. This situation can occur when using HPositionGenericInfo if the application has not allocated a large enough buffer.

aStatus returns KErrCancel if the request was successfully cancelled.

Panics:

The panic EPositionServerBadHandle occurs if no connection has been established to Location Server.
The panic EPositionDuplicateRequest occurs if the client already has an outstanding request to obtain position updates or to get the last known position (RPositioner::GetLastKnownPosition()) on the same RPositioner sub-session.

3.5.3.7 Canceling requests

Methods:

TInt CancelRequest(TInt aRequestId)

Usage:

This method attempts to cancel an outstanding asynchronous request from within the RPositioner class. Canceling requests is typically attempted when an application is closing down.

The aRequestId parameter specifies the request to be cancelled. For example, to cancel a call to NotifyPositionUpdate(), the parameter must be set to EPositionerNotifyPositionUpdate.

As CancelRequest() is an asynchronous method, Location Server typically queues the cancellation. Generally, Location Server attempts to abort the specified request shortly after the Cancel method has returned.
Return values:

KErrNone is returned if the Cancel request has been successfully sent to Location Server. However, this does not mean that the specified request has actually been terminated. This is indicated via the request status variable of the specified request.

Error codes:

KErrNotFound is returned if there is no outstanding request with the specified ID.

Also standard client-server error codes can be returned. See Section Error! Reference source not found. for more details.

Panics:

The panic EPositionServerBadHandle occurs if no connection has been established with Location Server.

3.5.4 TCoordinate

TCoordinate is used to hold the basic coordinates of a location (latitude, longitude and altitude). The class does not hold the accuracy of the location data or any time data. These are added by its derived classes TLocality (see Section Error! Reference source not found.) and TPosition (see Section Error! Reference source not found.).

As discussed in Section 3.3.1.7, the valid interval ranges for latitude and longitude are [-90, +90] and [-180, +180) respectively. If an application attempts to assign values that are outside these ranges to an instance of the TCoordinate class, the values are adjusted to be within the correct interval.

If only the longitude of a position is outside the correct interval, the adjustment is straightforward. For example, if an application attempts to set the position with latitude = +45 and longitude = +185, this is stored as latitude = +45 and longitude = -175.

In a case where the latitude is outside the correct interval, adjustments can be more complicated. In some instances, an adjustment of latitude may result in changing of longitude.

For example, assume an application gives the position latitude = +95 and longitude = +10, which is the point on Western hemisphere near the North Pole and Greenwich meridian. In this situation, both the latitude and longitude need to be adjusted: the correct latitude is +85 degrees, but this now gives a position on the opposite, i.e. Eastern, hemisphere. As the actual position lies in the Western hemisphere, the correct longitude is −170 degrees.

3.5.4.1 Constructors

Methods:

TCoordinate()
TCoordinate(const TReal64& aLatitude, const TReal64& aLongitude)
TCoordinate(const TReal64& aLatitude, const TReal64& aLongitude, TReal32 aAltitude)

Usage:
The constructors enable initial values for latitude, longitude and altitude to be set. If any of these values are not supplied, they default to NaN (see Section Error! Reference source not found.).

The datum of the co-ordinates defaults to WGS-84 (KPositionDatumWgs84) although this can be changed using the SetDatum() method.

The class assumes latitude in the range [-90, +90] and a longitude in the range [-180, +180). If values outside these ranges are specified, they are adjusted to be within these intervals. See Section 3.3.1.7 for further details.

3.5.4.2 Extracting the coordinates

Methods:

<table>
<thead>
<tr>
<th>TReal64 Latitude() const</th>
</tr>
</thead>
<tbody>
<tr>
<td>TReal64 Longitude() const</td>
</tr>
<tr>
<td>TReal32 Altitude() const</td>
</tr>
</tbody>
</table>

Usage:

These methods return the latitude, longitude and altitude of a point.

By default, a positioning module always returns horizontal positioning information, but does not guarantee to provide altitude.

If the application has explicitly chosen to accept partial updates, then any value may be returned unassigned. See Section Error! Reference source not found. for more details.

Return values:

Latitude() returns a degree value in the range [-90, +90] relative to the datum.

Longitude() returns a degree value in the range [-180, +180) relative to the datum.

Altitude() returns the height in meters relative to the current datum.

Error codes:

The value is set to NaN if the positioning module is not able to provide the information. See Section Error! Reference source not found. for more details.

3.5.4.3 Setting the coordinates

Methods:

<table>
<thead>
<tr>
<th>void SetCoordinate (const TReal64&amp; aLatitude, const TReal64&amp; aLongitude)</th>
</tr>
</thead>
<tbody>
<tr>
<td>void SetCoordinate (const TReal64&amp; aLatitude, const TReal64&amp; aLongitude, const TReal32 aAltitude)</td>
</tr>
</tbody>
</table>

Usage:

These methods assign the latitude, longitude and altitude to the TCoordinate class.
The first overload enables just the horizontal position to be specified. In this case, the altitude is set to NaN.

The class assumes latitude in the range [-90, +90] and longitude in the range [-180, +180). If values outside of these ranges are specified, they are adjusted to be within these intervals. See Section Error! Reference source not found. for further details.

3.5.4.4 Coordinate datum

Methods:

```cpp
TPositionDatumId Datum() const
void SetDatum(TPositionDatumId& aDatum)
```
Usage:

These methods assign and retrieve the datum of the co-ordinates. When an instance of the class is constructed, the default value of KPositionDatumWgs84 (WGS-84) is assigned.

3.5.4.5 Calculating the distance

Methods:

TInt Distance(const TCoordinate& aCoordinate, TReal32& aDistance) const

Usage:

This method calculates the horizontal distance between the current position and the supplied point specified by aCoordinate.

The parameter aDistance returns the calculated distance in meters, which is always returned as a positive value.

Return values:

KErrNone is returned on success.

Error codes:

KErrArgument is returned if information in either instance is incomplete, e.g. if a part of the horizontal co-ordinate is unassigned.

3.5.4.6 Calculating the bearing

Methods:

TInt BearingTo(const TCoordinate& aTargetCoord, TReal32& aBearing) const

Usage:

This method calculates the horizontal bearing from the current point to the supplied target location (aTargetCoord).

The parameter aBearing returns the bearing to the target point in degrees relative to the true north.

Return Values:

KErrNone is returned on success.

Error Codes:

KErrArgument is returned if information in either instance is incomplete, e.g. if a part of the horizontal co-ordinate is unassigned.

KErrPositionIncalculable is returned if this coordinate is at a pole or if the two coordinates are the same or antipodal.

3.5.4.7 Moving a coordinate

Methods:
**Usage:**

This method translates the given coordinate by the specified distance and bearing. The bearing must be supplied in degrees and relative to the true north. The distance is specified in meters.

**Return values:**

KErrNone is returned on success.

**Error codes:**

KErrArgument is returned if information in the current instance is incomplete. This occurs if a part of the horizontal coordinate is unassigned.

KErrPositionIncalculable is returned if this coordinate is at a pole.

### 3.5.5 TLocality

This class is derived from TCoordinate and it adds an error estimate for the horizontal and vertical accuracy of the point. The accuracy information is held in a TReal32 and it is measured in meters. The class also provides its own methods for determining the distance and bearing to a target point. These methods also provide an error estimate.

This class is further derived by TPosition (see Section Error! Reference source not found.), which adds time information.

#### 3.5.5.1 Constructors

**Methods:**

```cpp
TLocality()  
TLocality(const TCoordinate& aCoordinate,     
          TReal32 aHorizontalAccuracy)  
TLocality(const TCoordinate& aCoordinate,     
          TReal32 aHorizontalAccuracy,    
          TReal32 aVerticalAccuracy)  
```

**Usage:**

The constructors enable initial values for the underlying coordinate and its associated horizontal and vertical accuracy to be set. If the coordinate and/or accuracy information is not supplied, they default to NaN (see Section Error! Reference source not found.).

**Panic:**

Attempting to supply a negative value for any accuracy parameter is invalid and causes the panic EPositionBadAccuracy.

#### 3.5.5.2 Setting and retrieving accuracy information

**Methods:**

```cpp
TInt Move(TReal32 aBearing, TReal32 aDistance)
```
Usage:

These methods assign and retrieve the associated accuracy of the coordinate information held in the parent class TCoordinate, see Section Error! Reference source not found.. The accuracy information is represented in meters.

SetHorizontalAccuracy() assigns the horizontal accuracy for the underlying coordinate. In particular, it does not reset or affect the vertical accuracy.

Similarly, SetVerticalAccuracy() assigns the vertical accuracy for the underlying coordinate. In particular, it does not reset or affect the horizontal accuracy.

SetAccuracy() is a convenience method that enables both the horizontal and vertical accuracy to be assigned in a single call.

Return values:

HorizontalAccuracy() returns an accuracy estimate in meters of the horizontal portion of the underlying coordinate. NaN is returned if no horizontal accuracy has been set.

VerticalAccuracy() returns an accuracy estimate in meters of the vertical portion of the underlying coordinate. NaN is returned if no vertical accuracy has been set.

Error codes:

Both methods return NaN if no corresponding value accuracy has been set (see Section Error! Reference source not found. for more details).

Panas:

Attempting to supply a negative value for any accuracy parameter is invalid and causes the panic EPositionBadAccuracy.

3.5.5.3 Calculating the distance

Methods:

TInt Distance
   (const TCoordinate& aCoordinate,
    TReal32& aDistance) const
TInt Distance
   (const TLocality& aLocality,
    TReal32& aDistance,
    TReal32& aDelta) const

Usage:

These methods calculate the horizontal distance between the current position and the supplied point specified as either TCoordinate (aCoordinate) or TLocality (aLocality).
The parameter \texttt{aDistance} returns the calculated distance in meters, which is always returned as a positive value.

When a \texttt{TLocality} parameter is supplied, the second overloaded method calculates an estimate of the accuracy of the calculated distance. The \texttt{aDelta} parameter returns this value in meters, which is derived from the horizontal accuracy information.

**Return values:**
Both methods return \texttt{KErrNone} on success.

**Error codes:**

\texttt{KErrArgument} is returned if information in either of the instances is incomplete, e.g. if a part of the horizontal coordinate is unassigned.

The distance with accuracy overload method also returns \texttt{KErrArgument} if the horizontal accuracy has not been assigned for either instance.

### 3.5.5.4 Calculating the bearing

**Methods:**

```cpp
TInt BearingTo
   (const TCoordinate& aTargetCoord, TReal32& aBearing) const

TInt BearingTo
   (const TLocality& aTargetLocality, TReal32& aBearing, TReal32& aDelta) const
```

**Usage:**

These methods calculate the horizontal bearing from the current point to the supplied target location.

The parameter \texttt{aBearing} returns the bearing to the target point in degrees.

The \texttt{aDelta} parameter returns an estimate of the accuracy of the calculated bearing in degrees. This is derived from the horizontal accuracy information in both instances.

**Return values:**

Both methods return \texttt{KErrNone} on success.

**Error codes:**

\texttt{KErrArgument} is returned if information in either of the instances is incomplete, e.g. if a part of the horizontal coordinate is unassigned.

The distance with accuracy overload method also returns \texttt{KErrArgument} if the horizontal accuracy has not been assigned for either instance.

\texttt{KErrPositionIncalculable} is returned:

- if the error circle (horizontal accuracy) of this locality includes a pole.
- if the two localities has overlapping error circles.
- if the error circle of this locality overlaps with the error circle of \texttt{aTargetLocality} when projected antipodal.
3.5.6 **TPosition**

This class is the standard data structure for retrieving location information. **TPosition** is derived from **TLocality** and subsequently the base class **TCoordinate**, see Section Error! Reference source not found.. **TPosition** adds a time dimension to the locality information. This enables the speed to be calculated from two **TPosition** instances.

The time reflects the universal time as obtained from the mobile terminal when the location fix was obtained. In the Symbian OS, universal time is also known as system time and it is equivalent to GMT. In particular, it does not indicate the time as obtained from the position technology (for example network or satellite time).

The time is contained in a **TTime** data structure that provides microsecond resolution. However, it should be noted that system clocks only provide a resolution of milliseconds or indeed hundredths of a second.

### 3.5.6.1 Constructors

#### Methods:

<table>
<thead>
<tr>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPosition()</td>
</tr>
<tr>
<td>TPosition(const TLocality&amp; aLocality, TTime aTime)</td>
</tr>
</tbody>
</table>

#### Usage:

The parameterized constructor initializes the base class with the parameter. **aTime** is used as the timestamp for the position.

### 3.5.6.2 Setting and retrieving the time

#### Methods:

<table>
<thead>
<tr>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>TTime Time() const</td>
</tr>
<tr>
<td>void SetTime(TTime&amp; aTime)</td>
</tr>
<tr>
<td>void SetCurrentTime()</td>
</tr>
</tbody>
</table>

#### Usage:

These methods assign and retrieve the time when the position fix was obtained.

**Time()** retrieves the time when the position fix was obtained. The value is in universal time.

**SetTime()** is used to timestamp a position fix. The value should reflect universal time.

**SetCurrentTime()** is a convenience method that sets the timestamp of the position to the current universal time.

**Return values:**

**Time()** returns the date and time as a number of microseconds since midnight, January 1st, 0 AD nominal Gregorian. The value is in universal time (GMT).

### 3.5.6.3 Calculating the speed

#### Methods:
TInt Speed(const TPosition& aPosition, TReal32& aSpeed) const
TInt Speed(const TPosition& aPosition, TReal32& aSpeed, TReal32& aDelta) const

Usage:

These methods calculate the horizontal speed between the current position and the supplied instance aPosition. The speed is calculated based on the coordinates and time associated with each instance.

The parameter aSpeed contains the calculated speed in meters per second, which is always returned as a positive value.

The second overload provides an estimate of the accuracy of the calculated speed. The aDelta parameter returns this value (in meters per second), which is derived from the accuracy information contained in the underlying TLocality class.

Small errors in the speed calculation may occur if the system is heavily loaded. For example, there can be a short delay in the positioning hardware producing the fix and the system time stamping it. In this situation, the timestamp in TPosition is slightly inaccurate.

Return values:

Both methods return KErrNone on success.

Error codes:

KErrArgument is returned if information in either of the instances is incomplete, e.g. if the time of the position fix has not been set or a part of the horizontal co-ordinate is unassigned. KErrArgument is also returned if time value in this position and in argument position is same.

The speed with accuracy overload method also returns KErrArgument if the horizontal accuracy has not been assigned for either instance.

3.5.7 TPositionInfo

This is the standard parameter type passed to RPositioner when a client application requests location information. It is a simple wrapper class used to hold the identifier of the positioning technology that produced the fix and a TPosition class that holds the location information itself.

3.5.7.1 Obtaining the module identifier and update type

Methods:

TPositionModuleId ModuleId() const
void SetModuleId(TPositionModuleId aModuleId)
TPosition UpdateType() const
void SetUpdateType(TPositionUpdateType aUpdateType)

Defined:

TPositionInfoBase (parent class).

Usage:
ModuleId() retrieves the identifier of the position module that generated the location fix. This identifier can then be used to retrieve details about the module. See Section Error! Reference source not found. for more details.

UpdateType() is supplied by the positioning module. It informs the client application of the type of the update it has been notified of. Currently, only one kind of update type is defined: EPositionUpdateGeneral. This method is designed to allow functionality to be added in the future.

SetModuleId() and SetUpdateType() are normally used only by the Location Server framework to assign the identifier of the module that obtained the location information.

Return values:

ModuleId() returns the identifier (TPositionModuleId) of the positioning module that obtained the position information.

In the current version of Mobile Location FW, the method UpdateType() always returns EPositionUpdateGeneral.

3.5.7.2 Getting and setting the position

Methods:

```c++
void GetPosition(TPosition& aPosition) const
void SetPosition(const TPosition& aPosition)
```

Usage:

These methods are used to store and retrieve the standard location information class (TPosition) for a fix.

GetPosition() copies the position information from the TPositionInfo wrapper class to the parameter aPosition.

In general, SetPosition() is not called by the client application. It is normally used by the Location Server framework to assign the position information.

3.5.8 HPositionGenericInfo

HPositionGenericInfo is derived from TPositionInfo, see Section Error! Reference source not found.. Its role is to provide a generic means of retrieving either common extended location information or data that is specific to a particular positioning technology. For example:

Speed is an example of common extended information. It can be provided by a number of different positioning technologies, but may not be provided by all of them.

Address details are an example of data that is specific to a particular positioning technology.

HPositionGenericInfo enables the client application to request a number of data fields. It is the responsibility of the positioning module to provide as much of the requested information as it can. Generic identifiers (short integers) are used to specify the data fields. The data fields themselves can be of most of the basic types and descriptors.

The client application is then able to check what fields the position module has supplied and retrieve their contents.
3.5.8.1 Heap usage and default sizes

Underlying the implementation of this class is a data buffer that is normally 1 kB in size. It is a Symbian OS convention that larger data constructs should be created on the heap and deleted after use. As HPositionGenericInfo has been designed to be a flat data structure, there is no need for it to be derived from CBase. Hence, it is classified as an H class and it can safely be derived from TPositionInfo.

The constructors take the default buffer size of 1 kB, which should be sufficient for most circumstances. However, it is possible to vary the size of the data buffer when an instance of the class is created. This may be required if a number of very long descriptors are to be transferred. An application may also choose to reduce the amount of used memory by selecting a smaller buffer size if it knows this will be appropriate.

By default, the maximum number of fields that can be requested and returned is 32. This value is specified by the second default parameter to the constructors. Information about each requested field occupies approximately 12 bytes. This does not include the actual data.

3.5.8.2 Module support

All position modules are required to support the use of HPositionGenericInfo. However, modules that can only provide the standard location information (as provided in Section Error! Reference source not found.) do not actually make use of its extra capabilities.

3.5.8.3 Standard field identifiers

Values from the enumeration TPositionFieldId are used to request individual fields via the HPositionGenericInfo class. The following table contains a complete list of the standard field names, their type and a brief description. Proprietary or module specific fields can be added after EPositionProprietaryFieldsBegin.

Most fields are grouped by capability. This refers to the set of capabilities of each positioning module as defined by the enumeration TPositionModuleInfo::TPositionCapability (see Section 3.5.13.3 for more details). For each capability, there are a number of associated fields. If a module reports that it has a particular capability, it is able to return at least some of these fields. For example, a module with the Speed capability is able to provide the current horizontal speed and may be able to supply vertical speed. See Section 3.5.13.3 for more details.

<table>
<thead>
<tr>
<th>Field ID</th>
<th>Value type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EPositionFieldNone</td>
<td></td>
<td>Reserved field ID. Used to terminate an array of requested fields.</td>
</tr>
<tr>
<td>EPositionFieldComment</td>
<td>TDes16</td>
<td>A free field that can be used for a comment. This field can be returned by any type of positioning technology.</td>
</tr>
<tr>
<td>Speed capability</td>
<td></td>
<td>Positioning technologies with the Speed capability return fields in this section.</td>
</tr>
<tr>
<td>EPositionFieldHori</td>
<td>TReal32</td>
<td>The returned value is in meters per</td>
</tr>
<tr>
<td>Field Name</td>
<td>Data Type</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>-----------</td>
<td>-------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>HorizontalSpeed</td>
<td>TReal32</td>
<td>The returned value is in meters per second.</td>
</tr>
<tr>
<td>VerticalSpeed</td>
<td>TReal32</td>
<td>The returned value is in meters per second.</td>
</tr>
<tr>
<td>VerticalSpeedError</td>
<td>TReal32</td>
<td>The returned value is in meters per second.</td>
</tr>
</tbody>
</table>

### Direction capability

Positioning technologies with the Direction capability return fields in this section.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heading</td>
<td>TReal32</td>
<td>The current instantaneous direction of traveling in degrees to the true north.</td>
</tr>
<tr>
<td>HeadingError</td>
<td>TReal32</td>
<td>Heading error in degrees.</td>
</tr>
</tbody>
</table>

### Compass capability

Positioning technologies with the Compass capability return fields in this section.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TrueCourse</td>
<td>TReal32</td>
<td>The current direction in degrees to the true north.</td>
</tr>
<tr>
<td>TrueCourseError</td>
<td>TReal32</td>
<td>The true course error in degrees.</td>
</tr>
<tr>
<td>MagneticCourse</td>
<td>TReal32</td>
<td>The current direction in degrees to the magnetic north.</td>
</tr>
<tr>
<td>MagneticVariation</td>
<td>TReal32</td>
<td>The magnetic course error in degrees.</td>
</tr>
</tbody>
</table>

### Address Capability

Positioning technologies with the Address capability return fields in this section.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Country</td>
<td>TDes16</td>
<td>The name of the country.</td>
</tr>
<tr>
<td>CountryCode</td>
<td>TDes16</td>
<td>Country as specified by the two letter ISO 3166-1 code.</td>
</tr>
<tr>
<td>State</td>
<td>TDes16</td>
<td>Indicates the state or province, etc.</td>
</tr>
<tr>
<td>City</td>
<td>TDes16</td>
<td>The name of the city.</td>
</tr>
<tr>
<td>District</td>
<td>TDes16</td>
<td>The municipal district. Generally, an administrative area within a city.</td>
</tr>
</tbody>
</table>
### Media Capability

**EPositionFieldMediaLinks**
- **Type**: TUint8
- **Description**: This field is used to request media link data. On return, it contains the number of links that have been provided by the positioning module.

**EPositionFieldMediaLinksStart**
- **Type**: TDes8
- **Description**: The first media link relevant to this location. The rest follow after this field. A media link is of the format: `Type/Format/URI`.

  - **Type** and **Format** are the standard major and minor MIME types of the media. **URI** provides the location of the media.
  
  **Examples**:
  - `text/html/http://www.symbian.com`
  
  If any part of the MIME type is not known, the corresponding field is empty.

### Building Capability

**EPositionFieldBuildingName**
- **Type**: TDes16
- **Description**: The name of the building.

**EPositionFieldBuildingFloor**
- **Type**: TDes16
- **Description**: Floor or level details within a building.
| EPositionFieldBuildingRoom     | TDes16 | Room name or number within the building. |
| EPositionFieldBuildingZone    | TDes16 | Section of a building. May also be used to indicate separate areas of a (large) room. |
| EPositionFieldBuildingTelephone | TDes16 | Telephone number associated with the location. |
| **NMEA Capability**          |        | Positioning technologies with the NMEA capability return the fields in this section. |
| EPositionFieldNMEA Sentences  | TUint8 | This field is used to request raw NMEA data. On return, it contains the number of sentences that have been provided by the positioning module. |
| EPositionFieldNMEA SentencesStart | TDes8 | The first NMEA sentence. The rest follow after this field. |
| **Satellite Capability**     |        | Positioning technologies with the Satellite capability return the fields in this section. |
| EPositionFieldSatelliteNumInView | TInt8 | The number of satellites currently in view. |
| EPositionFieldSatelliteNumUsed | TInt8 | The number of satellites being used to provide position information. |
| EPositionFieldSatelliteTime   | TTime  | The time as obtained from the satellites. |
| EPositionFieldSatelliteHorizontalDoP | TReal32 | Horizontal dilution of precision. |
| EPositionFieldSatelliteSeaLevelAltitude | TReal32 | Altitude above the mean sea level. |
| EPositionFieldSatelliteGeoidalSeparation | TReal32 | The difference between the WGS-84 earth ellipsoid and the mean sea level. A negative value indicates that the geoid is below the WGS84 ellipsoid. |
| **Custom Fields**            |        | Reserved for device specific or proprietary information fields. |
3.5.8.4 Constructors

Methods:

```c
HPositionGenericInfo* New
    (TInt aBufferSize = KPositionGenericInfoDefaultBufferSize,
     TInt aMaxFields = KPositionGenericInfoDefaultMaxFields)
HPositionGenericInfo* NewL
    (TInt aBufferSize = KPositionGenericInfoDefaultBufferSize,
     TInt aMaxFields = KPositionGenericInfoDefaultMaxFields)
HPositionGenericInfo* NewLC
    (TInt aBufferSize = KPositionGenericInfoDefaultBufferSize,
     TInt aMaxFields = KPositionGenericInfoDefaultMaxFields)
```

Usage:

In common with other heap based classes in the Symbian OS, instances of HPositionGenericInfo are created using the supplied (static) New() methods.

These methods enable the transfer buffer size and the maximum number of transferred fields to be varied if required. The standard buffer size is 1024 bytes and the default maximum number of fields is 32.

Return values:

All of the New() methods return a pointer to a newly created instance of the class on the heap.

In addition, NewLC() pushes a pointer to the newly created instance on to the clean-up-stack.

Error codes:

New() returns NULL if it was unable to allocate sufficient memory on the heap.

NewL() and NewLC() both leave with KErrNoMemory if it was not possible to allocate sufficient memory on the heap.

PANics:

The panic EPositionGenericInfoZeroBufferSize occurs if the parameter aBufferSize is zero.

The panic EPositionGenericInfoZeroFields occurs if the parameter aMaxFields is zero.

3.5.8.5 Selecting the requested fields

Methods:

```c
void ClearRequestedFields()
TInt SetRequestedField(TPositionFieldId aFieldId)
TInt SetRequestedFields(const TPositionFieldIdList aFieldIdList)
```

Usage:
ClearRequestedFields() deletes any previous lists of fields. This method should be employed if the same instance of HPositionGenericInfo is used to obtain a different list of fields.

SetRequestedField() adds the field specified in aFieldId to the list of the desired fields.

SetRequestedFields() takes the array of field identifiers in aFieldList and adds the identifiers to the list of the desired fields. Note: The array of field identifiers must be terminated by the element EPositionFieldNone.

Return values:

SetRequestedField() and SetRequestedFields() both return KErrNone if the specified field ID has been added to the desired list.

Error codes:

SetRequestedField() and SetRequestedFields() return KErrOverflow if the total number of desired fields exceeds the maximum number that was specified when the class was created.

If the aFieldIdList parameter in SetRequestFields() is not terminated with EPositionFieldNone, the effect is undefined. This typically results in an arbitrary panic, although in some circumstances the Location Acquisition API may return KErrOverflow.

3.5.8.6 Retrieving the buffer size and the maximum number of fields

Methods:

TInt BufferSize() const
TInt MaxFields() const

Usage:

These methods retrieve the size of the transfer buffer and the maximum number of fields that can be transferred for the instance of HPositionGenericInfo. These values have been set when the class was constructed.

Return values:

BufferSize() returns the size of the internal transfer buffer in bytes.

MaxFields() returns the maximum number of fields that can be requested by the client and transferred by Location Server.

3.5.8.7 Testing for the requested fields

Methods:

TBool IsRequestedField(TPositionFieldId aFieldId) const
TPositionFieldId FirstRequestedFieldId() const
TPositionFieldId NextRequestedFieldId(TPositionFieldId aFieldId) const

Usage:

IsRequestedField() determines if the field specified in aFieldId is in the list of the desired fields.
FirstRequestedFieldId() returns the first identifier in the sequence of requested fields. This should be regarded as an arbitrary starting point and not (necessarily) the field that was first added to the list. The returned value can then be used in conjunction with NextRequestedFieldId() to iterate through the current list of the desired fields.

NextRequestedFieldId() iterates through the remaining requested fields. It takes the identifier of a previously returned field and returns the next field in the sequence.

Return values:
IsRequestedField returns ETrue if the specified field has been requested, otherwise EFalse.
FirstRequestedFieldId() and NextRequestedFieldId() return either a field identifier (other than zero) or zero if there are no further fields in the list of the desired fields.

3.5.8.8 Returned fields

Methods:

```cpp
void ClearPositionData()
TBool IsFieldAvailable(TPositionFieldId aFieldId) const
```

Usage:
ClearPositionData() is generally used only by the positioning system infrastructure and not by a client side application. It clears all data that has been entered (using SetValue) into the generic transfer buffer.

IsFieldAvailable() determines if the field specified in aFieldId is contained in the transfer buffer returned by the positioning module.

Return values:
IsFieldAvailable() returns ETrue if the specified field is contained in the transfer buffer, otherwise EFalse. See Section 3.5.8.9 for details on how to retrieve the data.

3.5.8.9 Getting and setting field data

Method:

```cpp
template <class TType> inline TInt GetValue(TPositionFieldId aFieldId, TType& aValue) const
template <class TType> inline TInt SetValue(TPositionFieldId aFieldId, const TType& aValue)
```

Usage:
These methods are used to store and retrieve the information.

The parameter aValue can be of the type TInt8, TInt16, TInt32, TInt64, TUInt8, TUInt16, TUInt32, TReal32, TReal64, TTime, TTimeIntervalMicroSeconds, Des8, or Des16.

In addition, when a descriptor is retrieved using GetValue(), the client application may also supply a TPtrC16 (or TPtrC8 if appropriate). Note: Using a
TPtrC in a call to GetValue() simply points the descriptor to the contents of the buffer in the HPositionGenericInfo instance. The data becomes invalid if the buffer is changed in any way. This includes the object going out of scope or being reused to retrieve position information. In addition, any attempt by the client application to add extra information to the buffer may result in its internal contents being rearranged. This also invalidates any pointers previously returned by GetValue().

Return values:

SetValue() and GetValue return KErrNone on successful operation.

Error codes:

GetValue() returns KErrNotFound if the specified field has not been returned by the positioning module.

In case the parameter aValue is of the type TDes, GetValue() returns KErrOverflow if the supplied descriptor is too short to contain the requested field.

SetValue() returns KErrPositionBufferOverflow if the data contained in the parameter aValue cannot be added to the class due to the buffer being too small. In this event, Mobile Location FW indicates the problem by completing the client's request with KErrPositionBufferOverflow.

Note: If a buffer overflow occurs when SetValue() is replacing an existing field, the current value is deleted from the buffer. In this case, the buffer does not hold either the old value or the new value for the specified field.

Panics:

GetValue() panics with EPositionGenericInfoMismatchDataType if there is a mismatch in the data type of a field. GetValue() must use the same data type as that assigned by SetValue().

3.5.9 TCourse

The class is used to hold information about the current speed and direction of the device. It is generally used in conjunction with TPositionCourseInfo when a positioning technology is able to supply these details as part of its positioning information.

Methods:

TCourse()
TReal32 Speed() const
TReal32 Heading() const
TReal32 SpeedAccuracy() const
TReal32 HeadingAccuracy() const
void SetSpeed(TReal32 aSpeed)
void SetHeading(TReal32 aBearing)
void SetSpeedAccuracy(TReal32 aSpeedAccuracy)
void SetHeadingAccuracy(TReal32 aBearingAccuracy)

Usage:

These methods are used to set and retrieve information about the current speed and direction. TCourse() is a standard constructor.

Return values:
**Speed()** returns the current speed in meters per second.

**Heading()** returns the current direction in degrees.

**SpeedAccuracy()** returns an estimate of the accuracy of the current speed in meters per second.

**HeadingAccuracy()** returns an estimate of the accuracy of the current heading in degrees.

**Error codes:**

All methods return NaN if the positioning module is unable to provide the associated information.
Panics:

Attempting to supply a negative value for any accuracy parameter is invalid and causes the panic EPositionBadAccuracy.

3.5.10 TPositionCourseInfo

This class can be passed as a parameter to RPositioner::NotifyPositionUpdate() when a client application wishes to request information about its current course. It is a simple wrapper class and contains an instance of TCourse. When a call to NotifyPositionUpdate completes, the TCourse member variable holds the speed and heading information.

Before calling RPositioner::NotifyPositionUpdate(), the client application should ensure that the positioning module is able to support requests for course information. This is normally done in the RPositioner::Open() method by stipulating either the Speed and/or Bearing capabilities via the parameter defining criteria. Another way to do it is to check the positioning module information using the RPositionServer::GetModuleInfoByIndex() or RPositionServer::GetModuleInfoById() methods. See Sections 3.5.2.3, 3.5.3.1 and 3.5.13.3 for further details.

Methods:

TPositionCourseInfo()
void GetCourse(TCourse& aCourse) const
void SetCourse(const TCourse& aCourse)

Usage:

These methods are used to set and retrieve information about the current speed and direction when a position technology module is able to determine these values.

TPositionCourseInfo() is a standard constructor.

GetCourse() returns the current course in a TCourse class which contains speed and direction. Note: The Speed and Direction properties of the TCourse class return the value NaN if the positioning module is unable to provide the information.

SetCourse() assigns the current course.

3.5.11 TSatelliteData

This class is used to hold information about the observed satellites. It is generally used in conjunction with TPositionSatelliteInfo when a positioning technology is able to supply these details as part of its positioning information.

Methods:

TSatelliteData()
TInt SatelliteId() const
TReal32 Azimuth() const
TReal32 Elevation() const
TInt SignalStrength() const
TBool IsUsed() const
void SetSatelliteId(TInt aSatelliteId)
void SetAzimuth(TReal32 aAzimuth)
void SetElevation(TReal32 aElevation)
void SetSignalStrength(TInt aSignalStrength)
void SetIsUsed(TBool aIsUsed)

Usage:
These methods are used to set and retrieve information about a particular satellite. TSatelliteData() is a standard constructor.

Return values:
SatelliteId returns a number identifying the satellite. In GPS, this is the PRN.

Azimuth returns the angle of the satellite in degrees from the device relative to the geographic north.

Elevation returns the angle of the satellite in degrees from the horizon (relative to the true north).

SignalStrength is an indication of the signal strength in dB received from the satellite.

IsUsed returns ETrue if the satellite is one of those being used to determine the position.

Error codes:
The Azimuth and Elevation methods return NaN if the positioning module is unable to provide the associated information.

SatelliteId returns -1 if the position module is unable to determine its identity.

3.5.12 TPositionSatelliteInfo

This class can be passed as a parameter to RPositioner::NotifyPositionUpdate() when a client application wishes to request information that has been obtained from satellites. It is a wrapper class and contains an instance of TSatelliteData for each observed satellite. It also contains information that has been derived from those satellites being used to obtain a fix (such as satellite time).

Before calling RPositioner::NotifyPositionUpdate(), the client application should ensure that the positioning module is able to support requests for course information. This is normally done in the RPositioner::Open() method by stipulating the Satellite capability via the parameter defining criteria. See Section 3.5.13.3 for further details.

3.5.12.1 Obtaining satellite information

Methods:

TPositionSatelliteInfo() TTime SatelliteTime() const
TReal32 HorizontalDoP() const
TReal32 VerticalDoP() const
TReal32 TimeDoP() const
void SetSatelliteTime(TTime aTime)
void SetHorizontalDoP(TReal32 aDoPValue)
void SetVerticalDoP(TReal32 aDoPValue)
void SetTimeDoP(TReal32 aDoPValue)

Usage:
These methods are used to set and retrieve derived information from the
satellites in view. TPositionSatelliteInfo() is a standard constructor.

Return values:
SatelliteTime() returns the time as obtained from the satellite system.
HorizontalDoP(), VerticalDoP() and TimeDoP() return the value indicating
the associated dilution of precision which is associated with the accuracy of a fix.

Error codes:
SatelliteTime() returns 0 if this information cannot be supplied by the
positioning module.
HorizontalDoP(), VerticalDoP() and TimeDoP() return NaN if the
associated value cannot be supplied by the positioning module.

3.5.12.2 Obtaining information on the satellites in view

Methods:

TInt NumSatellitesInView() const
TInt NumSatellitesUsed() const
void ClearSatellitesInView()
TInt GetSatelliteData(TUint aIndex, TSatelliteData& aSatelliteData) const
TInt AppendSatelliteData(const TSatelliteData& aSatelliteData)

Usage:
These methods are used to store and retrieve information about the satellites
currently in view.

GetSatelliteData() is used by client applications to retrieve the details of a
satellite. The parameter aIndex must be in the range of 0 to
NumSatellitesInView() – 1.

AppendSatelliteData() is used by Mobile Location FW to store details of a
satellite. In addition, this method updates the count of the number of satellites
used if appropriate. The details of the maximum of KMaxSatellitesInView
satellites can be stored.

Return values:
NumSatellitesInView() returns the number of satellites on which the
positioning modules has information.

NumSatellitesUsed() returns the number of satellites being used to obtain a
position fix.

GetSatelliteData() returns KErrNone if aIndex specifies an observed
satellite.

AppendSatelliteData() returns KErrNone if the supplied satellite
information has been successfully stored.

Error codes:
GetSatelliteData() returns KErrNotFound if aIndex is outside the range of 0 to \(\{\text{NumSatellitesInView()} - 1\}\).

AppendSatelliteData() returns KErrOverflow if an attempt is made to store details of more than KMaxSatellitesInView.

### 3.5.13 TPositionModuleInfo

The class is used to provide the client application with information about a positioning module. This includes the type of technology used to obtain information (e.g. network or terminal based), the likely quality of the position obtained (e.g. accuracy) and also what information it will provide.

The list of the available modules and their details can be retrieved via the RPositionServer class. See Section Error! Reference source not found. for more details.

#### 3.5.13.1 TTechnologyType

The method TPositionModuleInfo::TechnologyType() returns a bit mask of values from the enumeration TTechnologyType. These values indicate the mechanisms and characteristics of the technology used to obtain position information.

<table>
<thead>
<tr>
<th>Enumeration</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETechnologyUnknown</td>
<td>0x0</td>
<td>Indicates that the positioning module is unaware of the technology used to obtain position information.</td>
</tr>
<tr>
<td>ETechnologyTerminal</td>
<td>0x1</td>
<td>If this flag is set, the primary positioning technology is handset based. For example, standard GPS.</td>
</tr>
<tr>
<td>ETechnologyNetwork</td>
<td>0x2</td>
<td>If this flag is set, the primary positioning technology is network based. For example, E-OTD.</td>
</tr>
<tr>
<td>ETechnologyAssisted</td>
<td>0x4</td>
<td>This flag indicates that the primary positioning mechanism receives assistance in some form. Generally to obtain a quicker or more accurate fix.</td>
</tr>
</tbody>
</table>

#### 3.5.13.2 TDeviceLocation

The method TPositionModuleInfo::DeviceLocation() returns a value from the enumeration TDeviceLocation. It is used to indicate that the positioning hardware is either external or integral to the device.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDeviceUnknown</td>
<td>Indicates that the positioning module is unaware of the hardware used to supply positioning information.</td>
</tr>
<tr>
<td>EDeviceInternal</td>
<td>This value indicates that the positioning hardware is integral to the terminal.</td>
</tr>
</tbody>
</table>
3.5.13.3 **TCapabilities**

The method `TPositionModuleInfo::Capabilities()` returns a bit mask of values from the enumeration `TCapabilities`. These values indicate the range of position information that the module is capable of returning.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECapabilityNone</td>
<td>0x000</td>
<td>An unassigned value.</td>
</tr>
<tr>
<td>ECapabilityHorizontal</td>
<td>$2^0$</td>
<td>Positioning modules with this capability support the <code>TPositionInfo</code> class and are able to provide latitude and longitude related information. See Section <strong>Error! Reference source not found.</strong> for more details.</td>
</tr>
<tr>
<td>ECapabilityVertical</td>
<td>$2^1$</td>
<td>Positioning modules with this capability support the <code>TPositionInfo</code> class and are able to provide height related information. See Section <strong>Error! Reference source not found.</strong> for more details.</td>
</tr>
<tr>
<td>ECapabilitySpeed</td>
<td>$2^2$</td>
<td>Positioning modules with this capability support the <code>TPositionCourseInfo</code> class and are able to provide information related to the current horizontal speed. See Section <strong>Error! Reference source not found.</strong> for more details. In addition, this capability indicates that the module may be able to supply the current vertical speed. The standard <code>TPositionCourseInfo</code> class does not support vertical speed, but both speeds can be obtained by using the standard generic buffer class <code>HPositionGenericInfo</code> (see Section <strong>Error! Reference source not found.</strong>). The relevant predefined fields are detailed in Section <strong>Error! Reference source not found.</strong>.</td>
</tr>
<tr>
<td>ECapabilityDirection</td>
<td>$2^3$</td>
<td>Positioning modules with this capability support the <code>TPositionCourseInfo</code> class and are able to provide heading related information. See Section <strong>Error! Reference source not found.</strong> for more details. This information can be obtained using the standard generic buffer class <code>HPositionGenericInfo</code> (see Section <strong>Error! Reference source not found.</strong>). The relevant predefined fields are detailed in Section <strong>Error! Reference source not found.</strong>.</td>
</tr>
<tr>
<td>Capability</td>
<td>Version</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>---------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ECapabilitySatellite</td>
<td>2^4</td>
<td>Positioning modules with this capability support the TPositionSatelliteInfo class. Such a module is able to return at least some satellite data. Applications must ensure that any returned value is valid before it is used. See section <strong>Error! Reference source not found.</strong> for more details. This information can be obtained using the standard generic buffer class HPositionGenericInfo (see Section <strong>Error! Reference source not found.</strong>). The relevant predefined fields are detailed in Section <strong>Error! Reference source not found.</strong>.</td>
</tr>
<tr>
<td>ECapabilityCompass</td>
<td>2^5</td>
<td>Positioning modules with this capability are able to return information related to a magnetic compass. This information can be obtained using the standard generic buffer class HPositionGenericInfo (see Section <strong>Error! Reference source not found.</strong>). The relevant predefined fields are detailed in Section <strong>Error! Reference source not found.</strong>.</td>
</tr>
<tr>
<td>ECapabilityNmea</td>
<td>2^6</td>
<td>Positioning modules with this capability are able to return location information using NMEA formatted text strings. This information can be obtained using the standard generic buffer class HPositionGenericInfo (see Section <strong>Error! Reference source not found.</strong>). The relevant predefined fields are detailed in Section <strong>Error! Reference source not found.</strong>.</td>
</tr>
<tr>
<td>ECapabilityAddress</td>
<td>2^7</td>
<td>Positioning modules with this capability are able to return information related to the postal address of the current location. This information can be obtained using the standard generic buffer class HPositionGenericInfo (see Section <strong>Error! Reference source not found.</strong>). The relevant predefined fields are detailed in Section <strong>Error! Reference source not found.</strong>.</td>
</tr>
<tr>
<td>ECapabilityBuilding</td>
<td>2^8</td>
<td>Positioning modules with this capability are able to return the current position in terms of where within a building it is. For example, this may include the floor and the room name. This information can be obtained using the standard generic buffer class HPositionGenericInfo (see Section <strong>Error! Reference source not found.</strong>). The relevant predefined fields are detailed in Section <strong>Error! Reference source not found.</strong>.</td>
</tr>
</tbody>
</table>
Positioning modules with this capability are able to provide a link to further information about the location. The standard mechanism is via a URL. The details of the link can be obtained using the generic buffer class HPositionGenericInfo (see Section Error! Reference source not found.). The relevant predefined fields are detailed in Section Error! Reference source not found..

### 3.5.13.4 TPositionClassFamily

The method TPositionModuleInfo::ClassesSupported() takes a parameter of the type TPositionClassFamily and returns the set of classes (from that family) which the module supports. For example, the value EPositionInfoFamily represents all the position information derived classes such as TPositionInfo and TPositionSatelliteInfo. The root of that family tree is the class TPositionInfoBase.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EPositionInfoFamily</td>
<td>All the classes supported by the positioning module that are derived from TPositionInfoBase. For example, TPositionInfo, TPositionCourseInfo and HPositionGenericInfo. When this value is passed to the ClassesSupported() method, it returns a bit mask of values corresponding to the enumeration TPositionInfoClassType.</td>
</tr>
<tr>
<td>EPositionModuleInfoFamily</td>
<td>All the classes supported by the positioning module that are derived from TPositionModuleInfoBase. For example, TPositionModuleInfo. When this value is passed to the ClassesSupported() method, it returns a bit mask of values corresponding to the enumeration TPositionModuleInfoClassType.</td>
</tr>
<tr>
<td>EPositionModuleStatusFamily</td>
<td>All the classes supported by the positioning module that are derived from TPositionModuleStatusBase. For example, TPositionModuleStatus. When this value is passed to the ClassesSupported() method, it returns a bit mask of values corresponding to the enumeration TPositionModuleStatusClassType.</td>
</tr>
<tr>
<td>EPositionModuleStatusEvent</td>
<td>All the classes supported by the positioning module that are derived from TPositionModuleStatusBase. For example, TPositionModuleStatus. When this value is passed to the ClassesSupported() method, it returns a bit mask of values corresponding to the enumeration TPositionModuleStatusClassType.</td>
</tr>
</tbody>
</table>
### Methods

Methods:

```cpp
TPositionModuleInfo()
TPositionModuleId ModuleId() const
TBool IsAvailable() const
void SetModuleName(TDesC const & aModuleName)
void GetModuleName(TDes & aModuleName) const
void GetPositionQuality(TPositionQuality const & aPosQuality) const
TTechnologyType TechnologyType() const
TDeviceLocation DeviceLocation() const
TPositionCapabilities Capabilities() const
TUint32 ClassesSupported(TPositionClassFamily aFamilyClass) const
TVersion Version() const
void SetModuleId(TPositionModuleId aModuleId)
void SetIsAvailable(TBool aIsAvailable)
void SetModuleName(const TDesC & aModuleName)
void SetPositionQuality(const TPositionQuality & aPosQuality)
void SetTechnologyType(TTechnologyType aTechnologyType)
void SetDeviceLocation(TDeviceLocation aDeviceLocation)
```
void SetCapabilities(TCapabilities aDeviceCapabilities)
void SetClassesSupported(TPositionClassFamily aClassType, TUint32 aSupportedClasses)
void SetVersion(TVersion aVersion)

Usage:

GetModuleName() supplies a displayable name for the module via the aModuleName parameter. This descriptor must be of the size KPositionMaxModuleName.

GetPositionQuality() returns quality information such as the expected accuracy via the aPosQuality parameter.

TPositionModuleInfo() is a standard constructor.

Return values:

ModuleId() returns a unique identifier that specifies the position module.

IsAvailable() indicates if the positioning module is available for use. In general, a module is unavailable if it has been taken off-line by the user via the control panel, or if there is a hardware problem.

TechnologyType() returns a bit mask of values indicating what are the primary technologies used to obtain position information. This can be a combination of terminal and/or network based mechanism. It also indicates if the primary mechanism has been assisted. See Section 3.5.13.1 for more details.

DeviceLocation() returns whether the positioning hardware is integral or external. See Section 3.5.13.2 for more details.

Capabilities() returns a bit mask of values indicating the range of information the positioning module is able to return. For example, capabilities include horizontal coordinates, height and speed. See Section 3.5.13.3 for more information.

ClassesSupported() takes a parameter of the type TPositionClassFamily and returns a bit mask of the supported classes from that family tree. For example, supplying the parameter EPositionInfoFamily returns a bit mask of values from the enumeration TPositionInfoClassType. The returned value indicates which position information classes are supported by the positioning module. In the current implementation, only the position information classes actually have a family tree of more than one class. See Section Error! Reference source not found. for more details.

Version() returns the version information of the positioning module.

Error codes:

TechnologyType() returns ETechnologyUnknown if the technology type information cannot be determined by the positioning module.

DeviceLocation() returns EDeviceUnknown if the device location information cannot be determined by the positioning module.

Capabilities() returns ECapabilityNone if the capabilities information cannot be determined by the positioning module.

ClassesSupported() returns zero if the parameter aFamilyClass does not represent a known value.

Panics:
ClassesSupported() panics with EPositionInvalidClassType if the aFamilyClass argument does not represent a known family member.

3.5.14 TPositionQuality

TPositionQuality is used to indicate the quality of information that a positioning module is able to provide. The class contains various access methods to obtain attributes such as speed, accuracy and cost. Each of the values returned by these methods represent the expected value for an attribute. For example, HorizontalAccuracy() returns the expected accuracy in meters of a location fix produced by that module.

An instance of the class is returned as a part of the module information accessed by one of the RPositionServer::GetModuleInfo methods. See Section Error! Reference source not found. for more details.

3.5.14.1 TCostIndicator

The method TPositionQuality::CostIndicator() returns a value from the enumeration TCostIndicator. It is used to indicate if there is likely to be a charge when obtaining location information via the associated positioning module.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECostUnknown</td>
<td>This is the unassigned value and should not be returned.</td>
</tr>
<tr>
<td>ECostZero</td>
<td>This value indicates that no cost is expected to be</td>
</tr>
<tr>
<td></td>
<td>incurred when obtaining a position fix.</td>
</tr>
<tr>
<td>ECostPossible</td>
<td>This value indicates that the positioning module is</td>
</tr>
<tr>
<td></td>
<td>uncertain if the user will incur a charge.</td>
</tr>
<tr>
<td>ECostCharge</td>
<td>This value indicates that the position module expects a</td>
</tr>
<tr>
<td></td>
<td>charge to be levied when obtaining position information.</td>
</tr>
</tbody>
</table>

3.5.14.2 TPow erConsumption

The method TPositionQuality::PowerConsumption() returns a value from the enumeration TPow erConsumption. It indicates via a simple scale the likely power consumption associated with using a particular positioning module.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EPowerUnknown</td>
<td>This value indicates that the positioning module is unable to determine the</td>
</tr>
<tr>
<td></td>
<td>likely power drain.</td>
</tr>
<tr>
<td>EPowerZero</td>
<td>This value indicates that no internal power will be used when obtaining a</td>
</tr>
<tr>
<td></td>
<td>position fix.</td>
</tr>
<tr>
<td>EPowerLow</td>
<td>This value indicates that the positioning module expects a minimum power</td>
</tr>
<tr>
<td></td>
<td>drain when using the associated technology. This may be comparable to the</td>
</tr>
<tr>
<td></td>
<td>power usage when the phone is in the Standby mode.</td>
</tr>
<tr>
<td>EPowerMedium</td>
<td>This value indicates that the positioning module expects a moderate power</td>
</tr>
<tr>
<td></td>
<td>drain when using the associated technology. This may be comparable to the</td>
</tr>
<tr>
<td></td>
<td>power usage</td>
</tr>
</tbody>
</table>
when the phone is being actively used.

| EPowerHigh | This value indicates that the positioning module expects a high power drain when using the associated technology. The use of this module will quickly consume the phone’s battery. |

3.5.14.3  Time taken to obtain a fix

Methods:

```cpp
TTimeIntervalMicroSeconds TimeToFirstFix() const
TTimeIntervalMicroSeconds TimeToNextFix() const
void SetTimeToFirstFix(TTimeIntervalMicroSeconds aTimeToFirstFix)
void SetTimeToNextFix(TTimeIntervalMicroSeconds aTimeToNextFix)
```

Usage:

Different positioning technologies can take a different about of time to obtain the first fix and subsequent fixes. For some technologies, these values can be the same, although satellite based mechanisms are likely to take longer to obtain their initial position.

The value of 0 (zero) indicates that no value has been assigned. This is the default for both of the time to fix attributes.

Return values:

TimeToFirstFix() returns the expected time to obtain the initial location fix.
TimeToNextFix() returns the expected time to obtain a fix when the current position is known.

Panics:

Attempting to supply a negative value for any time interval parameter is invalid and causes the panic EPositionBadTime.

3.5.14.4  Accuracy

Methods:

```cpp
TReal32 HorizontalAccuracy() const
TReal32 VerticalAccuracy() const
void SetHorizontalAccuracy(TReal32 aHorizontalAccuracy)
void SetVerticalAccuracy(TReal32 aVerticalAccuracy)
```

Usage:

Horizontal accuracy reflects the error for the latitude and longitude components of a position. Vertical accuracy is the error in the altitude component. Both horizontal and vertical values are specified in meters.

If no value is set, the default value for both parameters is NaN. See Section Error! Reference source not found. for more details. Note: The value of zero should not be used for this purpose.

Return values:
HorizontalAccuracy() returns the expected error in meters of the latitude and longitude components of a position.

VerticalAccuracy() returns the expected error in meters of the altitude component of a position.

Error codes:
The value of NaN indicates that no value has been specified. See Section Error! Reference source not found. for more details.

Panics:
Attempting to supply a negative accuracy is invalid and causes the panic EPositionBadAccuracy.

3.5.14.5 Cost and power

Methods:

| TCostIndicator CostIndicator() const |
| TPowerConsumption PowerConsumption() const |
| void SetCostIndicator(TCostIndicator aCost) |
| void SetPowerConsumption(TPowerConsumption aPower) |

Usage:
These methods are used to indicate the expected cost and power consumption of obtaining a position fix.

The power consumption is reported on a simple scale ranging from zero to High. It indicates the expected internal power usage involved in obtaining a fix.

The cost indicator attempts to indicate if there will be a monetary charge for obtaining a fix. It does not attempt to represent how much it will cost.

If no value is set, the default values are ECostUnknown for the cost indicator and EPowerUnknown for power consumption. If a positioning module is uncertain about the cost of obtaining a fix, it should use the value ECostPossible rather than ECostUnknown.

Return values:
CostIndicator() returns a value as defined by the enumeration TPositionQuality::TCostIndicator.

PowerConsumption() returns a value as defined by the enumeration TPositionQuality::TCostIndicator.

Error codes:
CostIndicator() returns TPositionQuality::ECostUnknown if no value has been set.

PowerConsumption() returns TPositionQuality::EPowerUnknown if no value has been assigned.

3.5.14.6 Comparing quality

Methods:
```cpp
TInt Compare(
    const TPositionQualityBase& aPositionQuality,
    TInt aElementToCompare,
    TPositionQualityItem::TResult& aComparison) const
TBool IsDefined(TInt aElementId) const
TInt HighWaterMark() const
```

**Defined:**

TPositionQualityBase (parent class)

**Usage:**

These methods are generally used by Mobile Location FW to compare the value of individual quality fields. The base class TPositionQuality holds the value of each quality parameter in a generic manner. Each quality parameter is held in a separate field and indexed by an integer. This enables values to be compared without knowing what they actually represent.

Compare() determines if the value indicated by aElementToCompare of the current instance has better, equal or worse quality than the corresponding value in the supplied parameter (aPositionQuality). On a successfully return, the parameter aComparison contains the outcome of the test.

The parameter aElementToCompare must be in the range of 0 to HighWaterMark(). However, it is possible that intervening fields are not assigned. The method IsDefined() is used to determine if a value is present in each instance before a comparison is made.

IsDefined() is used prior to Compare() to ensure that the specified field has actually been assigned a value.

HighWaterMark() returns the index of the highest numbered field that has been assigned.

**Return values:**

Compare() returns KErrNone on a successful comparison.

IsDefined() returns ETrue if the field referenced by aElementId has been assigned a value. Otherwise, returns EFalse otherwise.

HighWaterMark() returns the index of the highest numbered field that has been assigned.

**Error codes:**

Compare() returns KErrArgument if either of the values referenced by aElementToCompare are unassigned.

Compare() also returns KErrArgument if there is a miss-match in either the type of the values or which magnitude is preferable. The latter error code can arise if one instance indicates that small values are preferred while the other believes that larger values provide quality.

### 3.5.15 TPositionModuleStatus

This class is used to provide information about the current status of a positioning module. Information about a positioning module can be obtained via the RPositionServer::GetModuleStatus() method or as a part of the details returned by RPositionServer:: NotifyModuleStatusEvent().
3.5.15.1 **TDeviceStatus**

The method `TPositionModuleStatus::DeviceStatus()` returns a value from the enumeration `TDeviceStatus`. It is used to indicate hardware type events emanating from the positioning device.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDeviceUnknown</td>
<td>This is not a valid state and must never be reported.</td>
</tr>
<tr>
<td>EDeviceError</td>
<td>This state is used to indicate that there are problems when using the device. For example, there may be hardware errors. It should not be confused with complete loss of data quality (see <code>TDataQualityStatus</code> in Section Error! Reference source not found. below), which indicates that the device is functioning correctly, but is currently unable to obtain position information. The error state is reported if the device cannot be successfully brought on line. For example, the positioning module may have been unable to communicate with the device or it is not responding as expected.</td>
</tr>
<tr>
<td>EDeviceDisabled</td>
<td>Although the device may be working properly, it has been taken off line and is regarded as being unavailable to obtain position information. This is generally done by the user via the control panel. In this state, Mobile Location FW will not use the device.</td>
</tr>
<tr>
<td>EDeviceInactive</td>
<td>This state signifies that the device is not being used by Mobile Location FW. This typically occurs because there are no clients currently obtaining positions from it.</td>
</tr>
<tr>
<td>EDeviceInitialising</td>
<td>This is a transient state. The device is being brought out of the Inactive state but has not reached either the Ready or Stand by modes. The initializing state occurs when the positioning module is first selected to provide a client application with location information.</td>
</tr>
<tr>
<td>EDeviceStandBy</td>
<td>This state indicates that the device has entered the Sleep or Power save mode. This signifies that the device is online, but it is not actively retrieving position information. A device generally enters this mode when the next position update is not required for some time and it is more efficient to partially power down. <strong>Note:</strong> Not all positioning modules support this state, particularly when there is external hardware.</td>
</tr>
<tr>
<td>EDeviceReady</td>
<td>The positioning device is online and is ready to retrieve position information.</td>
</tr>
<tr>
<td>EDeviceActive</td>
<td>The positioning device is actively in the process of retrieving position information. <strong>Note:</strong> Not all positioning modules support this state, particularly when there is external hardware.</td>
</tr>
</tbody>
</table>
3.5.15.2 \textit{TDataQualityStatus}

The method \texttt{TPositionModuleStatus::DataQualityStatus()} returns a value from the enumeration \texttt{TDataQualityStatus}. It is used to indicate the accuracy of the data emanating from a positioning module.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{EDataQualityUnknown}</td>
<td>This is an unassigned valued. This state should only be reported during an event indicating that a positioning module has been removed.</td>
</tr>
<tr>
<td>\texttt{EDataQualityLoss}</td>
<td>This state indicates that the accuracy and contents of the position information has been completely compromised. It is no longer possible to return any coherent data. This situation occurs if the device has lost track of all the transmitters (for example, satellites or base stations). It should be noted that although it is currently not possible to obtain position information, the device may still be functioning correctly. This state should not be confused with a device error (see \texttt{TDeviceStatus} in Section Error! Reference source not found.).</td>
</tr>
<tr>
<td>\texttt{EDataQualityPartial}</td>
<td>The state signifies that there has been a partial degradation in the available position information. In particular, it is not possible to provide the required (or expected) quality of information. This situation may occur if the device has lost track of one of the transmitters (for example, satellites or base stations).</td>
</tr>
<tr>
<td>\texttt{EDataQualityNormal}</td>
<td>The positioning device is functioning as expected.</td>
</tr>
</tbody>
</table>

3.5.15.3 \textit{Methods}

Methods:

\begin{verbatim}
TPositionModuleStatus()
TDeviceStatus DeviceStatus() const
TDataQualityStatus DataQualityStatus() const
void SetDeviceStatus(TDeviceStatus aStatus) const
void SetDataQualityStatus(TDataQualityStatus aStatus) const
\end{verbatim}

Usage:

These methods are used to obtain the status of a module. \texttt{DeviceStatus()} provides information about the current state of the positioning hardware utilized by the module. \texttt{DataQualityStatus()} reports details about the accuracy of the information that can be obtained.

\texttt{TPositionModuleStatus()} is a standard constructor.

Return values:

\texttt{ModuleId()} returns a unique identifier that specifies the position module.

\texttt{DeviceStatus()} indicates the current condition of the device. This ranges from a hardware error, through the device being in the Standby mode to actively
retrieving a fix. See Section Error! Reference source not found. for more
details.

DataQualityStatus() indicates the current condition of the position
information. For example, it may be impossible to obtain any information. In
some cases only partial information can be available. See Section Error!
Reference source not found. for more details.

Error codes:

DeviceStatus() returns TPositionModuleStatus::EDeviceUnknown if the
device status cannot be determined by the positioning module.

DataQualityStatus() returns
TPositionModuleStatus::EDataQualityUnknown if the data quality status
cannot be determined by the positioning module.

3.5.16 TPositionModuleStatusEvent

This class is used in conjunction with
RPositionServer::NotifyModuleStatusEvent() to receive notification
when the status of a positioning module changes. It is possible for the client
application to be notified when a sub-set of the status information changes.

3.5.16.1 TModuleEvent

The enumeration TModuleEvent is used by the client application to indicate the
events that are of interest. When an event occurs, the method
OccurredEvents() returns the event(s) that have been triggered.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EEEventNone</td>
<td>This is an unassigned value and should not be reported or used.</td>
</tr>
<tr>
<td>EEEventDeviceStatus</td>
<td>Events about the general status of the device. When this type of an event</td>
</tr>
<tr>
<td></td>
<td>occurs, client applications should inspect the value returned by the</td>
</tr>
<tr>
<td></td>
<td>TPositionModuleInfo::DeviceStatus() method for more information. See Section</td>
</tr>
<tr>
<td></td>
<td>Error! Reference source not found. for more details.</td>
</tr>
<tr>
<td>EEEventDataQuality</td>
<td>Events about the quality of the data a module is able to return. When this</td>
</tr>
<tr>
<td>Status</td>
<td>type of an event occurs, client applications should inspect the value</td>
</tr>
<tr>
<td></td>
<td>returned by the TPositionModuleInfo::DataQualityStatus() method for more</td>
</tr>
<tr>
<td></td>
<td>information. See Section Error! Reference source not found. for more</td>
</tr>
<tr>
<td></td>
<td>details.</td>
</tr>
<tr>
<td>EEEventSystemModule</td>
<td>System level events about the status of modules. Events of this type</td>
</tr>
<tr>
<td>Event</td>
<td>indicate when modules have been added or removed from the system. When this</td>
</tr>
<tr>
<td></td>
<td>event type occurs, client applications should inspect the value returned</td>
</tr>
<tr>
<td></td>
<td>by TPositionModuleInfo::SystemModuleEvent() to determine which particular</td>
</tr>
<tr>
<td></td>
<td>event was responsible. See Sections 3.5.16.2 and Error! Reference source</td>
</tr>
<tr>
<td></td>
<td>not found. for more details.</td>
</tr>
</tbody>
</table>
3.5.16.2 **TSystemModuleEvent**

The method `TPositionModuleStatusEvent::SystemModuleEvent()` returns a value from the enumeration `TSystemModuleEvent`. It is used to indicate system level type events emanating from Mobile Location FW. See Section Error! Reference source not found. for more details.

**Note**: When a system level event is generated, the status of the module is undefined. In particular, the application should not inspect the status of the module with `TPositionModuleStatusEvent::GetModuleStatus()`.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESystemUnknown</td>
<td>This is not a valid state and should never be reported.</td>
</tr>
<tr>
<td>ESystemModuleError</td>
<td>This state is used to indicate that there are problems using the module. For example, the module may have terminated abnormally. It should not be confused with the module reporting the error EDeviceError via <code>TPositionModuleStatus::DeviceStatus()</code>. That signifies that the module itself is up and running, but it may be unable to successful communicate with the hardware.</td>
</tr>
<tr>
<td>ESystemModuleInstalled</td>
<td>This state indicates that a new positioning module has been dynamically added to the system. To receive this event, the client application must have expressed interest in status notifications from any positioning module. The ID of the newly installed module can be found by calling <code>TPositionModuleStatusEvent::ModuleId()</code>.</td>
</tr>
<tr>
<td>ESystemModuleRemoved</td>
<td>The EDeviceRemoved event is generated when a positioning module is uninstalled. The ID of the removed module can be found by calling <code>TPositionModuleStatusEvent::ModuleId()</code>.</td>
</tr>
</tbody>
</table>

3.5.16.3 **Constructors**

**Methods:**

```c++
TPositionModuleStatusEvent()
TPositionModuleStatusEvent(TModuleEvent aRequestedEventMask)
```

**Usage:**
The parameter `aRequestedEventMask` is a bit mask that enables the client application to specify that it wants to be notified on a sub-set of status changes. See Section Error! Reference source not found. for more details. If no events are specified on construction, then the default behavior is to be notified on all types of status changes. The client can change its requested events later using the method `SetRequestedEvents()`.

3.5.16.4 Requesting and obtaining event types

Methods:

```cpp
void SetRequestedEvents(TModuleEvent aRequestedEventMask)
TModuleEvent RequestedEvents() const
void SetOccurredEvents(TModuleEvent aOccurredEventMask)
TModuleEvent OccurredEvents() const
```

Defined:

TPositionModuleStatusEventBase (parent class)

Usage:

These methods are used to specify which type of events the client application is interested in and to return what events have occurred.

The client application can specify a sub-set of events that it is interested in either when the class is constructed or by using the `SetRequestedEvents()` method. These take a bit mask of the requested events. See Section Error! Reference source not found. for more details.

To obtain a status change notification, the client posts the request using the `RPositionServer::NotifyModuleStatusEvent` method. This completes when one of the requested status events occurs.

The client can determine what status events have been triggered by calling the `OccurredEvents()` method. This returns a bit mask of those events.

Return values:

- `RequestedEvents()` returns a bit mask of the status event categories in which a client application is interested. See Section Error! Reference source not found. for more details.
- `OccurredEvents()` returns a bit mask of which event categories have changed their state. This method should be called after the client’s `RPositionServer::NotifyModuleStatusEvent()` has successfully completed.

3.5.16.5 System level events

Methods:

```cpp
TSystemModuleEvent SystemModuleEvent() const
void SetSystemModuleEvent(TSystemModuleEvent aSystemModuleEvent)
```

Defined:

TPositionModuleStatusEventBase (parent class)

Usage:
In a case where `OccurredEvents()` reports that an event of the type `EEventSystemModuleEvent` has occurred, `SystemModuleEvent()` is used to determine which particular system event was responsible.

Set `SystemModuleEvent()` in generally only called by Mobile Location FW and not by client applications.

**Return Values:**

`SystemModuleEvent()` returns a value of the type `TSystemModuleEvent` which indicates which system level event has occurred.

### 3.5.16.6 Module ID

**Methods:**

```cpp
TPositionModuleId ModuleId() const
void SetModuleId(TPositionModuleId aModuleId)
```

**Defined:**

`TPositionModuleStatusEventBase` (parent class)

**Usage:**

`ModuleId()` returns the ID of the module that triggered the event.

To obtain a status change notification, the client posts a request using the `RPositionServer::NotifyModuleStatusEvent` method. This completes when one of the requested status events types occurs with `ModuleId()` holding the ID of the module that triggered the event.

**Return values:**

`ModuleId()` returns the ID of the module that triggered the event.

### 3.5.16.7 Retrieving the module status

**Methods:**

```cpp
void GetModuleStatus(TPositionModuleStatus& aModuleStatus) const
void SetModuleStatus(const TPositionModuleStatus& aModuleStatus)
```

**Usage:**

When a module based status event occurs, the `TPositionModuleStatusEvent::GetModuleStatus()` method is a convenience function that enables applications to determine the status of the module. It removes the need to make a separate call to `RPositioner::GetModuleStatus()`.

There are two types of module based status events that affect the module status: `EEventDeviceStatus` and `EEventDataQualityStatus`. Device status events indicate that the current state of the positioning hardware has changed. Quality status events are reported when the accuracy of the available information varies. See Sections Error! Reference source not found. and Error! Reference source not found. for more information.

When either of these event types are generated, `GetModuleStatus()` can be used to obtain the module status. However, this method should not be used
when the system level event EEventSystemModuleEvent occurs. That event type reports when modules are either installed or removed from the system.

The client can determine what status events have been triggered by calling the OccurredEvents() method. This returns a bit mask of those events.

### 3.5.17 TPositionUpdateOptions

This class is used to change the behavior of RPositioner::NotifyPositionUpdate(). It enables the client application to request periodic updates. It can also be used to inform the positioning module that the client application will accept partial (i.e. incomplete) position information. This can be useful in situations such as GPS where information on orbiting satellites can be supplied before the device’s location can be determined.

#### 3.5.17.1 Constructors

**Methods:**

```
TPositionUpdateOptions()
TPositionUpdateOptions( TTimeIntervalMicroSeconds aInterval, 
                        TTimeIntervalMicroSeconds aTimeOut = 0, 
                        TTimeIntervalMicroSeconds aMaxAge = 0, 
                        TBool aAcceptPartialUpdates = EFalse)
```

**Panics:**

Attempting to supply a negative value for any time interval parameters is invalid and causes the panic EPositionBadTime.

#### 3.5.17.2 Update interval

**Methods:**

```
TTimeIntervalMicroSeconds UpdateInterval() const
void SetUpdateInterval( TTimeIntervalMicroSeconds aInterval)
```
Usage:

SetUpdateInterval is used if the client application wishes to receive updates at regular intervals. The interval is specified by the parameter aInterval. After each update notification, the client application still needs to re-post the NotifyPositionUpdate request. However, Mobile Location FW attempts to provide the client application with a fixed interval between updates by taking into account how long it has been since it supplied the previous notification.

To invoke periodic updates, the client application must call RPositioner::SetUpdateOptions(). The system may then attempt to synchronize the client application with the constraints of the positioning technology.

If the client has a notification request outstanding when SetUpdateOptions() is called, the position update request completes as soon as possible. The next call to NotifyPositionUpdate() then completes after the specified time interval from that point.

If the client does not have an outstanding notification request posted when this update option is set, the first call to RPositioner::NotifyPositionUpdate() completes as soon as possible. Subsequent update notifications then occur after the specified interval time from that point.

As noted, it is the responsibility of the client application to re-issue the request after each notification. If this takes longer than the update time interval, the behavior is undefined. In this case, Mobile Location FW is at liberty to supply an update immediately or to wait until the next schedule update would have taken place.

If there is an out-of-memory condition, then RPositioner::NotifyPositionUpdate() may return immediately. Applications should take this into account for reducing unnecessary requests.

Return values:

UpdateInterval() returns the current periodic time that the client application has specified. Zero is returned if periodic updates have not been enabled.

PANICS:

Attempting to supply a negative value for the time interval is invalid and causes the panic EPositionBadTime.

3.5.17.3 Update Timeout

Methods:

```cpp
void SetUpdateTimeOut(TTimeIntervalMicroSeconds aTimeOut)
TTimeIntervalMicroSeconds UpdateTimeOut() const
```

Usage:

SetUpdateTimeOut() specifies the time period for how long the client application is prepared to wait to obtain a fix. The timeout period starts when the client application issues a NotifyPositionUpdate() request. If an update is not obtained within this time period, NotifyPositionUpdate() completes with KErrTimedOut.
The default behavior is to have no time limit. This can also be specified by calling SetUpdateTimeOut() with the value of 0 (zero).

**Notes:**

If a request to obtain location information results in the user being asked to authorize it, the time taken for the user to respond is counted. When the application sets a timeout period, it should calculate for this possibility. See Section *Error! Reference source not found.* for more details.

When using a timeout, the application should also be aware of the time taken to obtain the initial fix, or the time taken to obtain a subsequent fix. These values can be obtained by calling one of the RPositionServer (see Section *Error! Reference source not found.*) RPositionServer::GetModuleInfo() method.

**Return values:**

UpdateTimeOut() returns the current timeout that the client application has specified. Zero is returned if no timeout limit has been set.

**Panics:**

Attempting to supply a negative value for the time interval is invalid and causes the panic EPositionBadTime.

### 3.5.17.4 Information age limit

**Methods:**

```cpp
TimeIntervalMicroSeconds MaxUpdateAge() const
void SetMaxUpdateAge(TimeIntervalMicroSeconds aMaxAge)
```

**Usage:**

SetMaxUpdateAge() is used in conjunction with SetUpdateInterval(). It specifies a time limit on the acceptable age for the information contained in a position update as returned by NotifyPositionUpdate(). It has no effect on the value returned by GetLastKnownLocation().

A typical age limit is the range of a few seconds. However, it must be less than the periodic update interval. Setting an age limit enables a positioning module to use a recently cached value rather than always instigating a new location fix. This enables greater efficiency, especially when there are multiple client applications.

The default value for the age limit is 0 (zero). This indicates that the positioning module is not allowed to return a cached value when the update interval is reached. Generally a new fix would be instigated.

**Return values:**

MaxUpdateAge() returns the current age limit for information returned by NotifyPositionUpdate(). Zero is returned when there is no acceptable age limit.

**Panics:**

Attempting to supply a negative value for the maximum age is invalid and causes the panic EPositionBadTime.
3.5.17.5  Partial updates

Methods:

TBool AcceptPartialUpdates() const
void SetAcceptPartialUpdates(TBool aAcceptPartialUpdates)

Usage:

In a normal operation, NotifyPositionUpdate() does not complete until the positioning module has obtained as much of the requested information as it is capable of obtaining. This always includes the standard position information contained in TPosition (see Section Error! Reference source not found.) and its parent classes.

Calling SetAcceptPartialUpdates with the value ETrue informs the module that the client application is willing to accept incomplete location information. It is then the responsibility of the client application to check the returned values. If partial updates are allowed, the TPosition class must include only a timestamp and other information can be missing. If partial updates are not allowed, the TPosition class must include latitude, longitude and a timestamp. Note that no other information is required to be returned. See Sections Error! Reference source not found. and Error! Reference source not found. for more details.

An example of where this can be useful is in GPS where information on the observed satellites is available before the device’s location can be determined. Requesting partial updates can enable the position module to pass information back as it becomes available. Here, the client application may receive a number of updates before a fix is obtained.

Partial updates can be used in conjunction with periodic update intervals (see SetUpdateInterval()). In this situation, the client application receives an update at the requested time, but the information may be incomplete.

Return values:

AcceptPartialUpdates returns ETrue if the positioning module has been informed to return partial location information. Otherwise, EFalse is returned.

3.5.18  CRequestor

The elements of the array type RRequestorStack contain pointers to the class CRequestor. Each instance of the CRequestor class is used to hold the identity of one of the parties involved in requesting the location. The class contains three data fields that indicate:

- If the requesting party is a service or an actual person (contact).
- A descriptor that identifies the requestor.
- In which format the information is, e.g. a telephone number, URL or email address.

3.5.18.1  TRequestorType

Values of this enumeration are used in SetRequestorL() to indicate if the requestor is a person or an application/service.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### ER equestorUnknown
This is an unassigned value and should not be reported or used.

### ER equestorService
This value is used to indicate that the request for location information originates from an application, network services or other entity. In particular, the requestor is not a person.

### ER equestorContact
This value is used to indicate that the request for location information originates from a person.

### 3.5.18.2 TRequestorFormat

Values of this enumeration are used in the `GetRequestorL()` method to identify the format of the `aData` parameter.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFormatUnknown</td>
<td>This is an unassigned value and should not be reported or used.</td>
</tr>
<tr>
<td>EFormatApplication</td>
<td>Indicates that <code>aData</code> contains the textual name of an application. Used only if the requestor type is <code>ERequestorService</code>.</td>
</tr>
<tr>
<td>EFormatTelephone</td>
<td>Indicates that <code>aData</code> contains a telephone number. This should be in the international format without spaces.</td>
</tr>
<tr>
<td>EFormatUrl</td>
<td>Indicates that <code>aData</code> contains an address in the URL format (as described in RFC2396).</td>
</tr>
<tr>
<td>EFormatMail</td>
<td>Indicates that <code>aData</code> contains an email address in the standard format used for internet mail (as described in RFC822).</td>
</tr>
</tbody>
</table>

### 3.5.18.3 Constructors

#### Methods:

```c
CRequestor* New(TRequestorType aType, TRequestorFormat aFormat, const TDesC& aData)
CRequestor* NewL(TRequestorType aType, TRequestorFormat aFormat, const TDesC& aData)
CRequestor* NewLC(TRequestorType aType, TRequestorFormat aFormat, const TDesC& aData)
CRequestor* NewL(RReadStream& aStream)
```

#### Usage:

These methods are used to construct instances of the class as common two-phased constructors conventional for Symbian OS. The first three methods allow the initialization of a newly created object with all the requestor’s data right at construction. The last variant reads the requestor data from a stream. A requestor object can be externalized to a stream by the `CRequestor::ExternalizeL()` method.
3.5.18.4 Setting and retrieving requestor details

Methods:

```cpp
void SetRequestorL(TRequestorType aType, TRequestorFormat aFormat, const TDesC& aData)
TRequestorType RequestorType() const
TRequestorFormat RequestorFormat() const
void GetRequestor(TRequestorType& aType, TRequestorFormat& aFormat, TPtrC& aData) const
```

Defined:

CRequestorBase (parent class)

Usage:

SetRequestorL() can be used to change the details of one of the parties requesting information.

The GetRequestor() method retrieves the details of one of the requesting parties.

Return values:

RequestorType() returns whether the requestor is a service or a contact. See Section 3.5.18.1 for more details.

RequestorFormat() returns the format in which the identify of the requestor is held, e.g. a telephone number or email address. See Section 3.5.18.2 for more details.

Error codes:

SetRequestorL() may leave with KErrNoMemory if it is unable to allocate enough memory to store the information.

3.5.18.5 Externalize and internalize

Methods:

```cpp
void InternalizeL(RReadStream& aStream)
void ExternalizeL(RWriteStream& aStream) const
```

Usage:

Client applications do not generally use these methods. They are normally used when data is being sent between the client application and Positioning Server.

InternalizeL() reads the supplied stream and resets the classes’ member data accordingly.

ExternalizeL() stores the current member data values to the supplied stream.

Error codes:

Both functions may leave with KErrNoMemory if there is insufficient heap memory to perform the operation.
3.5.19  RRrequestorStack

There are two versions of the RPositioner::SetRequestor() method. One version takes an RRequestorStack parameter and is typically used when there are applications or services involved in forwarding the requesting location information.

Most standard applications do not use RRequestorStack. Instead, they call the simpler RPositioner::SerRequestor() method to identify themselves.

RRequestorStack is typically used only if the application needs to identify a chain of requestors, e.g. if a remote party is requesting the location and this is routed through a local application. In this situation, the application must identify both itself and the remote party.

RRequestorStack is derived from the standard RPointerArray class and is defined to have elements of the type CRequestor*. This array type provides the standard set of methods for adding and retrieving items from the array. See RPointerArray in reference document 0 for more details. In addition, RRequestorStack adds its own methods used to read and write its contents to a stream.

3.5.19.1 Order of requestors

RRequestorStack is used to hold the identities of the initial, originating requestor and any gateway, service or application that was involved in forwarding the request.

The first item (element zero) on the requestor stack is the identity of the originating requestor. The last item on the stack is the application on the Symbian OS platform that finally interacts with Positioning Server.

If any other application, service or gateway was involved in forwarding the request, their identities are placed on the stack in the order in which they were encountered after the request left the originating requestor.

3.5.19.2  Externalize and internalize

Methods:

```cpp
void InternalizeL(RReadStream& aStream)
void ExternalizeL(RWriteStream& aStream) const
```
Usage:

Client applications do not generally use these methods. Normally they are used when data is being sent between the client application and Positioning Server.

InternalizeL() reads the supplied stream and resets the classes’ member data accordingly.

ExternalizeL() stores the current member data values to the supplied stream.

Error codes:

InternalizeL() and ExternalizeL() leave with KErrNoMemory if there is insufficient heap memory to perform the operation.

Panics:

ExternalizeL() panics with the code EPositionNullRequestor if there is a NULL pointer to a requestor in the array. This will happen if the client application has added a NULL pointer to the array rather than a pointer to an instance of CRequestor.

3.6 Code architecture

In order to use the Location Acquisition API, applications should:

- Include the system header file lbs.H.
- Link against the library: lbs.DLL (lbs.LIB)

The error, leave and panic codes defined in the Location Acquisition API can be found from LbsErrors.H.

The HPositionGenericInfo predefined field identifiers can be found in LbsFieldIds.H. The position class types are defined in LbsClassTypes.H.