



# **Programmer's Reference for Garmin iQue 3600 Handheld**

® Copyright 2004 PalmSource and Garmin Ltd. or its subsidiaries. All Rights Reserved. This documentation may be printed and copied solely for use in developing products for the iQue 3600 handheld. In addition, two (2) copies of this documentation may be made for archival and backup purposes. Except for the foregoing, no part of this documentation may be reproduced or transmitted in any form or by any means or used to make any derivative work (such as translation, transformation or adaptation) without express written consent from Garmin Ltd.

Garmin Ltd. reserves the right to revise this documentation and to make changes in content from time to time without obligation on the part of Garmin Ltd. to provide notification of such revision or changes.

GARMIN LTD. AND ITS SUPPLIERS MAKE NO REPRESENTATIONS OR WARRANTIES THAT THE DOCUMENTATION IS FREE OF ERRORS OR THAT THE DOCUMENTATION IS SUITABLE FOR YOUR USE. THE DOCUMENTATION IS PROVIDED ON AN “AS IS” BASIS. GARMIN LTD. AND ITS SUBSIDIARIES AND SUPPLIERS MAKE NO WARRANTIES, TERMS OR CONDITIONS, EXPRESS OR IMPLIED, EITHER IN FACT OR BY OPERATION OF LAW, STATUTORY OR OTHERWISE, INCLUDING WARRANTIES, TERMS, OR CONDITIONS OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND/OR SATISFACTORY QUALITY. TO THE FULL EXTENT ALLOWED BY LAW, GARMIN LTD. ALSO EXCLUDES FOR ITSELF, ITS SUBSIDIARIES, AND ITS SUPPLIERS ANY LIABILITY, WHETHER BASED IN CONTRACT OR TORT (INCLUDING NEGLIGENCE), FOR DIRECT, INCIDENTAL, CONSEQUENTIAL, INDIRECT, SPECIAL, OR PUNITIVE DAMAGES OF ANY KIND, OR FOR LOSS OF REVENUE OR PROFITS, LOSS OF BUSINESS, LOSS OF INFORMATION OR DATA, OR OTHER FINANCIAL LOSS ARISING OUT OF OR IN CONNECTION WITH THIS DOCUMENTATION, EVEN IF GARMIN, LTD., ITS SUBSIDIARIES, OR ITS SUPPLIERS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Palm OS, the Palm logo, PalmSource, Graffiti 2, HotSync, Palm, Palm Powered, the Palm Powered logo, the PalmSource logo, and the HotSync logo are trademarks of PalmSource, Inc.

Garmin® is a registered trademark and iQue™ and Que™ are trademarks of Garmin Ltd. or its subsidiaries and may not be used without the express permission of Garmin.

IF THIS DOCUMENTATION IS PROVIDED ON A COMPACT DISC, THE OTHER SOFTWARE AND DOCUMENTATION ON THE COMPACT DISC ARE SUBJECT TO THE LICENSE AGREEMENT ACCOMPANYING THE COMPACT DISC.

# Table of Contents

---

<b>Overview</b>	<b>1</b>
Purpose of This Document.....	1
Conventions Used in This Document .....	1
Tools for Software Development .....	1
Garmin SDK.....	2
<b>Features</b>	<b>3</b>
Feature Creator.....	3
Feature Numbers .....	3
<b>GPS Library</b>	<b>5</b>
Introduction to the GPS Library.....	5
GPS Library Data Structures.....	6
GPS Library Constants.....	11
GPS Library Functions.....	12
<b>Pen Input Manager</b>	<b>17</b>
Introduction to the Pen Input Manager .....	17
Pen Input Manager Data Structures .....	19
Pen Input Manager Constants .....	21
Pen Input Manager Functions .....	21
<b>Additional Hardware Buttons</b>	<b>25</b>
Introduction to the Additional Buttons .....	25
Button Activity Reporting.....	26
Button Constants .....	26
Responding to the Additional Buttons .....	26
<b>Power Manager Library</b>	<b>29</b>
Introduction to the Power Manager Library .....	29
Power Manager Library Functions .....	30
<b>Que API Library</b>	<b>33</b>
Introduction to the Que API Library.....	33
Que API Library Data Structures.....	39
Que API Library Constants.....	41
Que API Library Functions.....	42



# Overview

---

## Purpose of This Document

*Programmer's Reference for Garmin iQue 3600 Handheld* is a part of the Garmin Software Development Kit. This document details the information necessary for software development for the Garmin iQue™ 3600 handheld, software Release 3 or later.

## Conventions Used in This Document

Throughout this document, a `fixed width font` is used to signify code elements such as files, functions, structures, fields, and bitfields.

## Tools for Software Development

### CodeWarrior for Palm OS® Platform

This contains the Integrated Development Environment (IDE) and all the tools required to develop Palm OS® applications. The development of the applications for the iQue 3600 was performed using CodeWarrior for Palm OS® Platform version 8.3. For more information, visit the Metrowerks web site at <http://www.metrowerks.com>.

### Palm OS® 5.0 SDK

For basic development information for Palm OS® applications, including the Palm OS® 5.0 SDK, visit <http://www.palmos.com>.

### Palm OS® 5.2 Simulator

This simulates a Palm OS® 5.2 device. It allows for the testing and debugging of applications. This may be found at <http://www.palmos.com/dev/tools/simulator/index.html>. This SDK is compatible only with the Palm OS 5.2 Simulator.

# Garmin SDK

## Components

GarminSimulator.zip includes new PalmSim.exe and DAL.dll files, as well as other DLL files to implement the Garmin extensions. It also includes in the AutoLoad folder the necessary PRCs for the Garmin extensions, as well as prebuilt PRCs for the Garmin examples GPSInfo.prc and PINMgrExample.prc.

GarminExamples.zip contains the source code for the Garmin examples. It also includes the prebuilt PRC for the third Garmin example, PwrMgrExample, since this application cannot be used in the simulator.

GarminSupport.zip contains the Garmin-specific include files.

## Unpacking the SDK

1. If you have not done so already, get the Palm OS® 5.2 debug simulator (Palm\_OS\_52\_Simulator\_Dbg.zip) from <http://www.palmos.com/dev/tools/simulator/index.html>, and unzip that onto your hard drive.
2. Copy the Palm OS® 5.2 Simulator "Debug" folder and all of its contents to a new folder named "GarminDebug".
3. Extract GarminSimulator.zip into this new "GarminDebug" folder.
4. Extract GarminExamples.zip into a convenient folder, such as the "(CodeWarrior Examples)" folder of your CodeWarrior installation.
5. Extract the GarminSupport.zip file into a convenient folder, such as the "Other SDKs" folder of your CodeWarrior installation. This will create a "Garmin" folder under the folder it is extracted into. Remember to add this folder to the access paths of any projects that need to use the Garmin-specific include files.
6. The first time you run the simulator, confirm that the RAM size Memory Setting is at least 32 MB and that the Dynamic Heap Size Memory Setting is at least 2048 KB.

# Features

---

The `FtrGet()` API may be used to determine if a feature is present in the Garmin handheld device. For more information on the `FtrGet()` API, see the Palm OS® documentation.

This chapter describes the various features available in the Garmin iQue 3600 handheld, which are defined in the header file `Garmin.h`. It discusses the following topics:

- Feature Creator
- Feature Numbers

## Feature Creator

To access the features unique to the Garmin Handheld, use `garminFtrCreator` as the `creator` argument for `FtrGet()`.

## Feature Numbers

For the `featureNum` argument, specify a value described below.

### **garminFtrNumPenInputServices**

Call `FtrGet()` with this value to determine if the Pen Input Manager API is present.

```
err = FtrGet( garminFtrCreator,
              garminFtrNumPenInputServices, &PenData
            );
```

If `ftrErrNoSuchFeature` is returned, the Pen Input Manager API is not present.

### **garminFtrNumExtraKeys**

Call `FtrGet()` with this value to determine if the additional hardware buttons are present.

```
err = FtrGet( garminFtrCreator,
              garminFtrNumExtraKeys, &KeyData );
```

If `ftrErrNoSuchFeature` is returned, the additional hardware buttons are not present.

## **garminFtrNumIntegratedGPS**

Call `FtrGet ( )` with this value to determine if an integrated GPS is present in the handheld.

```
err = FtrGet( garminFtrCreator,
              garminFtrNumIntegratedGPS, &GPSdata );
```

If `ftrErrNoSuchFeature` is returned, an integrated GPS is not present in the handheld.

## **garminFtrNumMedia**

Call `FtrGet ( )` with this value to determine the media features present in the handheld.

```
err = FtrGet( garminFtrCreator,
              garminFtrNumMedia, &MediaData );
```

If `ftrErrNoSuchFeature` is returned, there are no media features present in the handheld. Otherwise, the third parameter will contain a set of bits, which are a mask for the different media features that are present, as specified below.

Media Mask	Description
<code>garminMediaIntegratedMicrophone</code>	An integrated microphone is present in the handheld
<code>garminMediaWAVOutput</code>	WAV output is supported by the handheld
<code>garminMediaMP3Output</code>	MP3 output is supported by the handheld



# GPS Library

---

To begin learning more about GPS, visit <http://www.garmin.com/aboutGPS>.

This chapter describes the GPS Library declared in the header file `GPSTLib68K.h`. It discusses the following topics:

- Introduction to the GPS Library
- GPS Library Data Structures
- GPS Library Constants
- GPS Library Functions

## Introduction to the GPS Library

### Using the GPS Library

The GPS Library provides access to the data from the internal GPS. To get access to the GPS Library, `#include GPSTLib68K.h` in your application.

Before the GPS Library can be used, it must be found or loaded, using the standard Palm OS® paradigm:

```
/*-----
Find the GPS library. If not found, load it.
-----*/
error = SysLibFind( gpsLibName, &gGPSTLibRef);
if (error != errNone)
{
    error = SysLibLoad
        (
            gpsLibType,
            gpsLibCreator,
            &gGPSTLibRef
        );
    ErrFatalDisplayIf( (error != errNone),
        "can't load GPS Library" );
}
```

The GPS Library normally computes new data once a second. When data is computed, the GPS Library broadcasts the notification

sysNotifyGPSTDataEvent. Once your application has registered for this notification, it can call the GPSGet functions when this notification is received. The GPSGet functions can also be used strictly on a polling or as needed basis.

Once your application is done using the GPS Library (normally when the application stops), you should close and unload the library using the standard Palm OS® paradigm:

```
/*-----  
Close the library.  
-----*/  
err = GPSClose( gGPSLibRef );  
  
/*-----  
Unload the GPS Library.  
-----*/  
if ( err != gpsErrStillOpen )  
{  
    SysLibRemove( gGPSLibRef );  
}
```

## **GPS Data and the Palm OS® Simulator**

GPS data may be received when using the Palm OS® Simulator by following these steps:

1. Connect a recent model Garmin GPS to a PC serial port. The Serial Data Format on the Garmin GPS unit must be set to “Garmin”, which is the default setting.
2. Right-click in the Simulator and select Settings|Communication|Communication ports. Select the Cradle Communication Port and bind it to the COM port to which the Garmin GPS is connected.
3. Turn on the Garmin GPS and put the unit into Simulator mode. With the unit in Simulator mode, it is possible for you to adjust position, velocity, altitude, and track on the unit and have those changes reflected in the Palm OS® Simulator. If satellite signals are available at your PC, GPS information will also be present in the Palm OS® Simulator when the unit is operated normally.

## **GPS Library Data Structures**

### **GPSFixT8**

GPST8 defines the quality of the position computation. Based on the number of satellites being received and the availability of differential correction (such as WAAS), the position may be known in two dimensions (latitude and longitude) or three dimensions (latitude, longitude, and altitude).

```
typedef Int8 GPST8; enum
{
    gpsFixUnusable    = 0,
    gpsFixInvalid     = 1,
    gpsFix2D          = 2,
    gpsFix3D          = 3,
    gpsFix2DDiff      = 4,
    gpsFix3DDiff      = 5
};
```

### Value Descriptions

gpsFixUnusable	GPS failed integrity check.
gpsFixInvalid	GPS is invalid or unavailable.
gpsFix2D	Two dimensional position.
gpsFix3D	Three dimensional position.
gpsFix2DDiff	Two dimensional differential position.
gpsFix3DDiff	Three dimensional differential position.

## GPST8

GPST8 defines the modes for the GPS.

```
typedef Int8 GPST8; enum
{
    gpsModeOff        = 0,
    gpsModeNormal     = 1,
    gpsModeBatSaver   = 2,
    gpsModeSim        = 3,
    gpsModeExternal   = 4
};
```

### Value Descriptions

gpsModeOff	GPS is off.
gpsModeNormal	Continuous satellite tracking.

<code>gpsModeBatSaver</code>	Periodic satellite tracking to conserve battery power.
<code>gpsModeSim</code>	Simulated GPS information.
<code>gpsModeExternal</code>	External source of GPS information.

## GPSPositionDataType

`GPSPositionDataType` defines the position data returned by the GPS. The `GPSPositionDataType` uses integers to indicate latitude and longitude in semicircles, where  $2^{31}$  semicircles are equal to 180 degrees. North latitudes and East longitudes are indicated with positive numbers; South latitudes and West longitudes are indicated with negative numbers. The following formulas show how to convert between degrees and semicircles:

$$\text{degrees} = \text{semicircles} * ( 180 / 2^{31} )$$
$$\text{semicircles} = \text{degrees} * ( 2^{31} / 180 )$$

```
typedef struct
{
    Int32      lat;
    Int32      lon;
    float      altMSL;
    float      altWGS84;
} GPSPositionDataType;
```

### Field Descriptions

<code>lat</code>	Latitude component of the position in semicircles.
<code>lon</code>	Longitude component of the position in semicircles.
<code>altMSL</code>	Altitude above mean sea level component of the position in meters.
<code>altWGS84</code>	Altitude above WGS84 ellipsoid component of the position in meters.

## GPSPVTDataType

`GPSPVTDataType` combines the GPS data types into one structure.

```
typedef struct
{
    GPSTatusDataType      status;
```

```
GPSPositionDataType    position;  
GPSVelocityDataType    velocity;  
GPSTimeDataType        time;  
} GPSPVTDataType;
```

### Field Descriptions

status	GPS status.
position	GPS position.
velocity	GPS velocity.
time	GPS time.

## GPSSatDataType

GPSSatDataType defines the data for one satellite.

```
typedef struct  
{  
    UInt8      svid;  
    UInt8      status;  
    Int16      snr;  
    float      azimuth;  
    float      elevation;  
} GPSSatDataType;
```

### Field Descriptions

svid	The space vehicle identifier for the satellite.
status	The status bitfield the for satellite (see constants later).
snr	The satellite signal to noise ratio * 100 (dB Hz).
azimuth	The satellite azimuth (radians).
elevation	The satellite elevation (radians).

## GPSStatusDataType

GPSStatusDataType defines the status data reported by the GPS.

```
typedef struct  
{  
    GPSModeT8    mode;  
    GPSFixT8     fix;  
    UInt16       filler2;  
    float        epe;
```

```
float    eph;  
float    epv;  
} GPSStatusDataType;
```

### Field Descriptions

mode	GPS mode.
fix	GPS fix.
filler2	Alignment padding.
epe	The one-sigma estimated position error in meters.
eph	The one-sigma horizontal only estimated position error in meters.
epv	The one-sigma vertical only estimated position error in meters.

## GPSTimeDataType

GPSTimeDataType defines the time data returned by the GPS.

```
typedef struct  
{  
    UInt32    seconds;  
    UInt32    fracSeconds;  
} GPSTimeDataType;
```

### Field Descriptions

seconds	Seconds since midnight UTC.
fracSeconds	To determine the fractional seconds, divide the value in this field by $2^{32}$ .

## GPSVelocityDataType

GPSVelocityDataType defines the velocity data returned by the GPS. The individual East, North, and up components completely describe the velocity. The track and speed fields are provided for convenient access to the most commonly used application of GPS velocity.

```
typedef struct  
{  
    float    east;  
    float    north;  
    float    up;
```

```
float    track;  
float    speed;  
} GPSVelocityDataType;
```

### Field Descriptions

east	The East component of the velocity in meters per second.
north	The North component of the velocity in meters per second.
up	The upwards component of the velocity in meters per second.
track	The horizontal vector of the velocity in radians.
speed	The horizontal speed in meters per second.

## GPS Library Constants

### GPS Library Error Codes

gpsErrNone	No error.
gpsErrNotOpen	The GPS Library is not open.
gpsErrStillOpen	The GPS Library is still open.
gpsErrMemory	Not enough memory.
gpsErrNoData	No GPS data available.

### Extended Notification Information

The GPS Library broadcasts a `sysNotifyGPSDataEvent` when the GPS information changes. The `notifyDetailsP` of this notification is a `UInt32` (not a pointer to a `UInt32`) which contains one of the following extended notification information values indicating the reason for the notification.

gpsLocationChange	The GPS position has changed.
gpsStatusChange	The GPS status has changed.
gpsLostFix	The quality of the GPS position computation has become less than two dimensional.

<code>gpsSatDataChange</code>	The GPS satellite data has changed.
<code>gpsModeChange</code>	The GPS mode has changed.

## Satellite Status Bitfield Values

These define the bits in the status field of `GPSSatDataType`.

<code>gpsSatEphMask</code>	Ephemeris: 0 = no ephemeris, 1 = has ephemeris.
<code>gpsSatDifMask</code>	Differential: 0 = no differential correction, 1 = differential correction.
<code>gpsSatUsedMask</code>	Used in solution: 0 = no, 1 = yes.
<code>gpsSatRisingMask</code>	Satellite rising: 0 = no, 1 = yes.

# GPS Library Functions

## GPSClose

<b>Purpose</b>	Close the GPS Library.	
<b>Prototype</b>	<code>Err GPSClose( const UInt16 refNum )</code>	
<b>Parameters</b>	<code>-&gt; refNum</code>	Reference number for the library.
<b>Result</b>	<code>gpsErrNone</code>	No error.
	<code>gpsErrStillOpen</code>	Couldn't be closed because the library is still in use by other applications.
<b>Comments</b>	<p>Closes the GPS Library and disposes of the global data memory if required. Called by any application or library that's been using the GPS Library and is now finished with it.</p> <p>This should not be called if <code>GPSOpen</code> failed.</p> <p>If <code>gpsErrStillOpen</code> is returned, the calling app should not call <code>SysLibRemove</code>.</p>	

## GPSGetLibAPIVersion



<b>Purpose</b>	Get the GPS Library API version.	
<b>Prototype</b>	<code>UInt16 GPSGetLibAPIVersion ( const UInt16 refNum )</code>	
<b>Parameters</b>	<code>-&gt; refNum</code>	Reference number for the library.
<b>Result</b>	The API version of the library.	
<b>Comments</b>	Can be called without opening the GPS Library first.	

## GPSGetMaxSatellites

<b>Purpose</b>	Get the maximum number of satellites.	
<b>Prototype</b>	<code>UInt8 GPSGetMaxSatellites ( const UInt16 refNum )</code>	
<b>Parameters</b>	<code>-&gt; refNum</code>	Reference number for the library.
<b>Result</b>	Maximum number of satellites that are currently supported.	
<b>Comments</b>	The value returned by this routine should be used in the dynamic allocation of the array of satellites (GPSSatDataType).	

## GPSGetPosition

<b>Purpose</b>	Get current position data.	
<b>Prototype</b>	<code>Err GPSGetPosition( const UInt16 refNum, GPSPositionDataType *position )</code>	
<b>Parameters</b>	<code>-&gt; refNum</code>	Reference number for the library.
	<code>&lt;- position</code>	Contains the latest position from the GPS.
<b>Result</b>	<code>gpsErrNone</code>	No error.
	<code>gpsErrNotOpen</code>	The GPS Library is not open.
	<code>gpsErrNoData</code>	No data has been received for a period of time.

**Comments** If the return value is not `gpsErrNone` , the data should be considered invalid.

## **GPSPVT**

**Purpose** Get current position, velocity, and time data.

**Prototype** `Err GPSPVT( const UInt16 refNum,  
GPSData *pvt )`

**Parameters**

<code>-&gt; refNum</code>	Reference number for the library.
<code>&lt;- pvt</code>	Contains the latest position, velocity, and time data from the GPS.

**Result**

<code>gpsErrNone</code>	No error.
<code>gpsErrNotOpen</code>	The GPS Library is not open.
<code>gpsErrNoData</code>	No data has been received for a period of time.

**Comments** If the return value is not `gpsErrNone` , the data should be considered invalid.

If `pvt->status.fix` is equal to `gpsFixUnusable` or `gpsFixInvalid`, the rest of the data in the structure should be considered invalid.

## **GPSSatellites**

**Purpose** Get current satellite data.

**Prototype** `Err GPSSatellites( const UInt16 refNum,  
GPSData *sat )`

**Parameters**

<code>-&gt; refNum</code>	Reference number for the library.
<code>&lt;- sat</code>	Contains latest satellite information from the GPS. See the comments below.

**Result**

<code>gpsErrNone</code>	No error.
<code>gpsErrNotOpen</code>	The GPS Library is not open.

`gpsErrNoData`                      No data has been received for a period of time.

**Comments**    If the return value is not `gpsErrNone` , the data should be considered invalid.

The `sat` parameter must point to enough memory to hold the maximum number of satellites worth of satellite data.

## GPSTatus

**Purpose**        Get current status data.

**Prototype**    `Err GPSTatus( const UInt16 refNum,  
                          GPSTatusDataType *status )`

**Parameters**    `-> refNum`                      Reference number for the library.  
                     `<- status`                      Contains the latest status from the GPS.

**Result**        `gpsErrNone`                      No error.  
                     `gpsErrNotOpen`                      The GPS Library is not open.  
                     `gpsErrNoData`                      No data has been received for a period of time.

**Comments**    If the return value is not `gpsErrNone` , the data should be considered invalid.

## GPSTime

**Purpose**        Get current time data.

**Prototype**    `Err GPSTime( const UInt16 refNum,  
                          GPSTimeDataType *time )`

**Parameters**    `-> refNum`                      Reference number for the library.  
                     `<- time`                      Contains latest time data from the GPS.

**Result**        `gpsErrNone`                      No error.  
                     `gpsErrNotOpen`                      The GPS Library is not open.

`gpsErrNoData` No data has been received for a period of time.

**Comments** If the return value is not `gpsErrNone` , the data should be considered invalid.

## **GPSGetVelocity**

**Purpose** Get current velocity data.

**Prototype** `Err GPSGetVelocity( const UInt16 refNum,  
GPSVelocityDataType *velocity )`

**Parameters**

<code>-&gt; refNum</code>	Reference number for the library.
<code>&lt;- velocity</code>	Contains the latest velocity data from the GPS.

**Result**

<code>gpsErrNone</code>	No error.
<code>gpsErrNotOpen</code>	The GPS Library is not open.
<code>gpsErrNoData</code>	No data has been received for a period of time.

**Comments** If the return value is not `gpsErrNone` , the data should be considered invalid.

## **GPSOpen**

**Purpose** Opens the GPS Library.

**Prototype** `Err GPSOpen( const UInt16 refNum )`

**Parameters**

<code>-&gt; refNum</code>	Reference number for the library.
---------------------------	-----------------------------------

**Result**

<code>gpsErrNone</code>	No error.
<code>gpsErrMemory</code>	Not enough memory to open the library.

**Comments** Opens the GPS Library and prepares it for use. Called by any application or library that wants to use the services that the library provides.

---

GPSSOpen must be called before calling any other GPS Library functions, with the exception of GPSGetLibAPIVersion.

# 4

## Pen Input Manager

---

This chapter describes the Pen Input Manager API declared in the header file `PenInputMgr.h`. It discusses the following topics:

- Introduction to the Pen Input Manager
- Pen Input Manager Data Structures
- Pen Input Manager Constants
- Pen Input Manager Functions.

### Introduction to the Pen Input Manager

#### Pen Input Manager

The Pen Input Manager controls the area of the screen that is traditionally silkscreened onto the device. On the iQue 3600, this area is controlled by software, and it is sometimes referred to as "soft graffiti" or "collapsible graffiti". This area is comprised of two parts. The upper part is the dynamic input area, or graffiti area; the lower part is the status bar. The dynamic input area can be open (shown) or closed (hidden), while the status bar is always shown.

There is a button in the status bar that allows the user to show or hide the dynamic input area. This button is called the "input trigger". It shows a down arrow if the dynamic input area is open, or an up arrow if the dynamic input area is closed.

The input trigger can be enabled or disabled. If the input trigger is enabled, the user can control the state of the dynamic input area; if the input trigger is disabled, the input trigger is grayed out and the user cannot control the state of the dynamic input area.

#### Dynamic Input Area Concepts

Normally, users are the ones who change the dynamic input area state by tapping the input trigger button in the status bar, but applications also have the ability to set the dynamic input area state and to disable the trigger that allows the user to change the state.

## Pen Input Manager

### *Introduction to the Pen Input Manager*

---

There are two dynamic input area states, open and closed. The function `PINSetInputAreaState()` changes the state of the dynamic input area. Applications may query the dynamic input area state using `PINGetInputAreaState()`.

There are two input trigger states, enabled and disabled. The function `PINSetInputTriggerState()` changes the state of the input trigger. Applications may query the input trigger state using `PINGetInputTriggerState()`.

There are two dynamic input area policies. The default is to have the dynamic input area open and the input trigger disabled. The second policy allows the application and the user to control the dynamic input area state and the input trigger state. Applications should set the form's dynamic input area policy by calling `FrmSetDIAPolicyAttr()` in the `frmLoadEvent`. Each form in an application will use the default policy if `FrmSetDIAPolicyAttr()` is not called by the application.

Applications should register what size they want to be in the `frmLoadEvent` by calling `WinSetConstraintsSize()`.

## Pen Input Manager Feature

The Pen Input Manager registers its API version with the feature manager. Use the following feature manager call to determine the Pen Input Manager API version:

```
err = FtrGet( pinCreator, pinFtrAPIVersion,
              &APIVersion );
```

The current Pen Input Manager API version is 1.0, and is fully compatible with the PalmSource™ Pen Input Manager API version 1.0.

If `FtrGet` returns `ftrErrNoSuchFeature`, then the Pen Input Manager is not present and should not be used.

## Using the Pen Input Manager

To get access to the Pen Input Manager, `#include PenInputMgr.h` in your 68K application. Since the Pen Input Manager is an extension and not a library, it is available without being found or loaded.

To enable the input trigger and therefore give users the ability to close the dynamic input area, you must make the following calls in the `frmLoadEvent`:

```
/*-----  
Set the constraints.
```

```
-----*/
WinSetConstraintsSize( WinGetDisplayWindow(),
    160, 160, pinMaxConstraintSize, 160, 160,
    160 );

/*-----
Set the dynamic input area policy.
-----*/
FrmSetDIAPolicyAttr( FrmGetActiveForm(),
    FrmDIAPolicyCustom );

/*-----
Enable the input trigger.
-----*/
PINSetInputTriggerState
    ( pinInputTriggerEnabled );
```

## **Determining When the Dynamic Input Area State Changes**

Whenever the state of the dynamic input area changes, the Pen Input Manager broadcasts a `sysNotifyDisplayResizedEvent`. Register for this notification if your application needs to know when the dynamic input area changes. If you register, be sure to unregister before your application exits. If you fail to unregister, "the system will crash when the notification is broadcast" (according to the *Palm OS® Programmer's Companion*).

## **Determining the Size of the Application Display Area**

`WinGetDisplayExtent()` returns the current size of the display window. Typically, at initialization and upon receipt of a `sysNotifyDisplayResizedEvent` notification, your application will get the current size of the display window and adjust the locations of the various user interface items as needed.

The supplied `PINMgrExample` application is provided to demonstrate the usage of various aspects of the Pen Input Manager.

# **Pen Input Manager Data Structures**

## **FrmDIAPolicyT16**

`FrmDIAPolicyT16` specifies the dynamic input area policy type.

```
typedef UInt16 FrmDIAPolicyT16; enum
```

```
{  
    frmDIAPolicyStayOpen,  
    frmDIAPolicyCustom  
};
```

#### **Value Descriptions**

frmDIAPolicyStayOpen	The dynamic input area stays open and the input trigger is disabled. This is the default.
frmDIAPolicyCustom	The dynamic input area state and input trigger state may be controlled by the application and the user.

## **PinInputAreaStateT16**

PinInputAreaStateT16 specifies the dynamic input area state.

```
typedef UInt16 PinInputAreaStateT16; enum  
{  
    pinInputAreaOpen,  
    pinInputAreaClosed,  
    pinInputAreaNone  
};
```

---

#### **Value Descriptions**

pinInputAreaOpen	The dynamic input area is displayed. This is the default.
pinInputAreaClosed	The dynamic input area is not being displayed.
pinInputAreaNone	There is no dynamic input area.

## **PinInputTriggerStateT16**

PinInputTriggerStateT16 specifies the input trigger state.

```
typedef UInt16 PinInputTriggerStateT16; enum  
{  
    pinInputTriggerEnabled,  
    pinInputTriggerDisabled,  
    pinInputTriggerNone  
};
```

#### **Value Descriptions**



<code>pinInputTriggerEnabled</code>	The status bar icon is enabled, meaning that the user is allowed to open and close the dynamic input area.
<code>pinInputTriggerDisabled</code>	The status bar icon is disabled, meaning that the user is not allowed to open and close the dynamic input area. This is the default.
<code>pinInputTriggerNone</code>	There is no dynamic input area.

## Pen Input Manager Constants

<code>pinMaxConstraintSize</code>	Maximum size for setting constraint sizes.
<code>pinErrInvalidParam</code>	An invalid state parameter was entered.

## Pen Input Manager Functions

### **FrmGetDIAPolicyAttr**

<b>Purpose</b>	Get a form's dynamic input area policy.
<b>Prototype</b>	<code>FrmDIAPolicyT16 FrmGetDIAPolicyAttr ( FormPtr formP )</code>
<b>Parameters</b>	<code>-&gt; formP</code> Pointer to a form.
<b>Result</b>	The form's dynamic input area policy.
<b>Comments</b>	This routine is used to determine a form's dynamic input area policy. The default dynamic input area policy is <code>frmDIAPolicyStayOpen</code> .

### **FrmSetDIAPolicyAttr**

<b>Purpose</b>	Set a form's dynamic input area policy.
<b>Prototype</b>	<code>Err FrmSetDIAPolicyAttr( FomrPtr formP, const FrmDIAPolicyT16 diaPolicy )</code>

## Pen Input Manager

### *Pen Input Manager Functions*

---

<b>Parameters</b>	-> formP	Pointer to a form.
	-> diaPolicy	The policy to use for this form.
<b>Result</b>	errNone	No error.
	pinErrInvalidParam	Parameter is not valid.

**Comments** This routine is used to set a form's dynamic input area policy, which will be used for opening and closing the dynamic input area. Applications should call this function in response to the frmLoadEvent. If an application does not call this function, the policy for that application will be frmDIAPolicyStayOpen.

## PINGetInputAreaState

**Purpose** Get the current state of the dynamic input area.

**Prototype** PinInputAreaStateT16 PINGetInputAreaState(void)

**Parameters** None

**Result** Current state of the dynamic input area.

**Comments** Call this routine to determine whether the dynamic input area is open or closed.

## PINGetInputTriggerState

**Purpose** Get the current state of the input trigger.

**Prototype** PinInputTriggerStateT16  
PINGetInputTriggerState( void )

**Parameters** None

**Result** Current state of the input trigger.

**Comments** Call this routine to determine if the input trigger is enabled or disabled.

## PINSetInputAreaState

**Purpose** Set the state of the dynamic input area.

<b>Prototype</b>	<pre>Err PINSetInputAreaState ( const PinInputTriggerStateT16 state )</pre>	
<b>Parameters</b>	-> state	The desired state of the dynamic input area.
<b>Result</b>	errNone	No error .
	pinErrInvalidParam	Parameter is not valid .
<b>Comments</b>	<p>This routine allows the application to set the state of the dynamic input area. Unless the appropriate constraints have been registered and the dynamic input area policy set to custom, the only state allowed is open.</p>	

## PINSetInputTriggerState

<b>Purpose</b>	Set the state of the input trigger.	
<b>Prototype</b>	<pre>Err PINSetInputTriggerState ( const PinInputTriggerStateT16 state )</pre>	
<b>Parameters</b>	-> state	The desired state of the input trigger.
<b>Result</b>	errNone	No error.
	pinErrInvalidParam	Parameter is not valid.
<b>Comments</b>	<p>This routine enables or disables the input trigger. Unless the appropriate constraints have been registered and the dynamic input area policy set to custom, the only state allowed is disabled.</p> <p>Normally, the trigger should remain enabled, allowing the user the choice of displaying the dynamic input area or not. In certain circumstances, an application might want to prevent the display of the dynamic input area or ensure the display of the dynamic input area. If the application disables the trigger, it should enable it in response to the appStopEvent.</p>	

## WinSetConstraintSize

<b>Purpose</b>	Register an application's size constraints.	
<b>Prototype</b>	<pre>Err WinSetConstraintsSize( WinHandle winHandle, const Coord minHeight, const Coord prefHeight, const Coord maxHeight, const Coord minWidth, const Coord prefWidth, const Coord maxWidth )</pre>	

## Pen Input Manager

### *Pen Input Manager Functions*

---

<b>Parameters</b>	-> <code>winHandle</code>	Handle to a window.
	-> <code>minHeight</code>	The minimum height to which this window can be sized.
	-> <code>prefHeight</code>	The preferred height for this window.
	-> <code>maxHeight</code>	The maximum height for this window.
	-> <code>minWidth</code>	The minimum width for this window.
	-> <code>prefWidth</code>	The preferred width for this window.
	-> <code>maxWidth</code>	The maximum width for this window.

<b>Result</b>	<code>errNone</code>	No error.
---------------	----------------------	-----------

**Comments** The values are specified using the standard coordinate system, which refers to the original screen size of 160 × 160.

Currently only the `maxHeight` parameter is used. If your application desires to allow the dynamic input area to be closed, specify the constant `pinMaxConstraintSize` for this parameter.

# Additional Hardware Buttons

---

This chapter describes the additional hardware buttons on the Garmin iQue 3600 Handheld. It discusses the following topics:

- Introduction to the Additional Buttons
- Button Activity Reporting
- Button Constants
- Responding to the Additional Buttons

## Introduction to the Additional Buttons

### Additional Buttons

To help provide support for one-hand applications, additional hardware buttons have been added to the side of the Garmin iQue3600.

The additional Garmin buttons are:

- a Thumbwheel, which can be pressed up, down, or in;
- an Escape button;
- a Record button.

To access these additional hardware buttons, `#include GarminChars.h` in your application.

### Garmin Buttons and the Palm OS® Simulator

The Garmin buttons have been mapped to keys in the supplied Palm OS® Simulator as follows:

Thumb Wheel Up:	F6
Thumb Wheel Down:	F8
Thumb Wheel In:	F7
Escape Button:	F9
Record Button:	F11

The Escape and Record button exhibit the "momentarily pressed" and "pressed and held" behavior described below.

## Button Activity Reporting

Button activity is reported by `keyDownEvents`. The Escape and Record buttons generate different data depending on whether they are momentarily pressed or pressed and held. If they are momentarily pressed, the `keyDownEvent` is sent when they are released. If they are pressed and held, the `keyDownEvent` is sent after they have been held for a period of time, even if the button has not been released.

The Garmin virtual character codes are sent in the `keyCode` field of the `keyDownEvent` data. The `keyDownEvents` also provide values in the `chr` field, to allow unmodified applications to respond to the additional buttons.

The Thumbwheel can also be held in. This action is dedicated to marking a waypoint at the current GPS position, and is not accessible to third-party developers.

## Button Constants

The values sent in the `keyCode` and `chr` fields are defined as follows:

Button	keyCode	chr
Thumbwheel up	<code>vchrGarminThumbWheelUp</code>	<code>vchrPageUp</code>
Thumbwheel down	<code>vchrGarminThumbWheelDown</code>	<code>vchrPageDown</code>
Thumbwheel in	<code>vchrGarminThumbWheelIn</code>	<code>chrCarriageReturn</code>
Escape	<code>vchrGarminEscape</code>	<code>vchrGarminEscape</code>
Escape held	<code>vchrGarminEscapeHeld</code>	<code>vchrGarminEscapeHeld</code>
Record	<code>vchrGarminRecord</code>	<code>vchrGarminRecord</code>
Record held	<code>vchrGarminRecordHeld</code>	<code>vchrGarminRecordHeld</code>

The values returned by `KeyCurrentState()` for Garmin keys are as follows:

Button	Value
Thumbwheel up	<code>keyBitGarminThumbWheelUp</code>
Thumbwheel down	<code>keyBitGarminThumbWheelDown</code>
Thumbwheel in	<code>keyBitGarminThumbWheelIn</code>
Escape	<code>keyBitGarminEscape</code>
Record	<code>keyBitGarminRecord</code>

## Responding to the Additional Buttons

Typically your application will respond to Garmin buttons by checking for a Garmin `keyDownEvent` before dispatching the event to any other handlers.

```
do
{
/*-----
Get an event.
-----*/
EvtGetEvent(&event, evtWaitForever);

/*-----
Send to each handler in order, if not
already used.
-----*/
if ( ! GarminKeyHandleEvent( &event ) )
{
    if ( ! SysHandleEvent( &event ) )
    {
        if ( ! MenuHandleEvent( 0, &event,
                                &error ) )
        {
            if ( ! AppHandleEvent( &event ) )
            {
                FrmDispatchEvent( &event );
            }
        }
    }
}
} while ( event.eType != appStopEvent );
```

You should not wait to handle the Garmin button event in `AppHandleEvent`, since the event contains values in the `chr` field and will likely be handled by the system or menu event handler.

The macro `GarminKeyIsGarmin()` in `GarminChars.h` can be used to detect if the `keyDownEvent` is one of the Garmin keys. If you process the event you should not dispatch it to the other event handlers, since the event contains values in the `chr` field and will likely also be handled by the system or menu event handler.





# Power Manager Library

---

This chapter describes the Power Manager Library declared in the header file `PwrMgrLib68K.h`. It discusses the following topics:

- Introduction to the Power Manager Library
- Power Manager Library Functions

## Introduction to the Power Manager Library

### Low Power Mode

When the iQue 3600 enters low power mode, the display and backlight are turned off, while the processor, GPS, and audio continue to operate normally. Low power mode can be used to extend the battery life while continuing to allow the handheld to execute applications, such as an audio player. If any application has enabled low power mode, when the auto-off time has expired the iQue 3600 will enter low power mode instead of powering off. Low power mode is indicated by the LED blinking briefly approximately every 10 seconds. Note that low power mode uses substantially more battery power than allowing the handheld to power off.

If an application desires to have the iQue 3600 enter low power mode when the auto-off time has expired, the application should enable low power mode. Low power mode will stay enabled until your application disables it; therefore it is **extremely** important that your application disables low power mode when it no longer needs to be enabled. Note that if your application **disables** low power mode it does not guarantee the handheld will power off, as another application could also have low power mode enabled; however if your application enables low power mode, it **does** guarantee that the handheld will enter low power mode and will **not** power off.

### Using the Power Manager Library

## Power Manager Library

### *Power Manager Library Functions*

---

The Power Manager Library provides access to this power saving functionality in the iQue 3600. To get access to the Power Manager Library, `#include PwrMgrLib68K.h` in your application.

Before the Power Manager Library can be used, it must be found or loaded, using the standard Palm OS® paradigm:

```
/*-----  
Find the Power Manager library. If not found,  
load it.  
-----*/  
error = SysLibFind( kPwrMgrLibName,  
    &gPwrMgrLibRef );  
if ( error != errNone )  
{  
    error = SysLibLoad( kPwrMgrLibType,  
        kPwrMgrLibCreator, gPwrMgrLibRef );  
  
    ErrFatalDisplayIf( (error != errNone),  
        "can't load Power Manager Library" );  
}
```

Once your application is done using the Power Manager Library (normally when the application stops), you should unload the library using the standard Palm OS® paradigm:

```
/*-----  
Remove the library.  
-----*/  
SysLibRemove( gPwrMgrLibRef );
```

The supplied `PwrMgrExample` application is provided to demonstrate the usage of the Power Manager Library.

## The Power Manager Library and the Palm OS® Simulator

The Power Manager Library is not supported by the Palm OS® Simulator.

## Power Manager Library Functions

### **PwrSetLowPowerMode**

**Purpose** Set low power mode.

**Prototype**    `Boolean PwrSetLowPowerMode( UInt16 refNum,  
                                      const UInt32 creator, const Boolean enable )`

**Parameters**

-> refNum	Reference number for the library.
-> creator	Creator ID of the calling application.
-> enable	Set low power mode to true or false.

**Result**    Returns true if the action was successful.

**Comments**    If enable is true, the handheld will enter low power mode when the auto-off time has expired. If enable is false, low power mode for your application is disabled.



# Que API Library

---

This chapter describes the Que API declared in the header file `QueAPI.h`. It discusses the following topics:

- Introduction to the Que API Library
- Que API Library Data Structures
- Que API Library Constants
- Que API Library Functions

## Introduction to the Que API Library

### Que API Library

The Que API Library provides access to Garmin map data stored in device's internal memory or stored on an external card. The Que API library allows applications to create points at a specified latitude and longitude, at the location of an address, at the location the user selects from a map, and at the location of an item the user selects through the find menu. The Que API library also allows applications to get information about a point, display a form showing the details of a point including its location on a map, display the map application centered on the point, and create a route from the current location to a point.

### Que API Library Concepts

The data returned when a point is created is a **handle** to the point, not the actual data for the point. The advantages to this approach include:

- Isolates applications from memory management issues.
- Isolates applications from the details of the point data structure and size, which helps ensure future compatibility.

Point handles can either be **open** or **closed**. A handle is **open** when it is associated with data for a point; handles are opened when a point is created. A handle is **closed** when it is not associated with data for a point. Calling `QueClosePoint()` closes open handles; closed handles have a value of `queInvalidPointHandle`.

The use of handles requires following a few simple rules:

- Before a handle is used it is considered closed; therefore handles must be initialized to `QueInvalidPointHandle`.
- A handle is opened when it is assigned a value from one of the following APIs:
  - `QueCreatePoint()`
  - `QueCreatePointFromEvent()`
  - `QueDeserializePoint()`
- Before your application exits, or when you are through using a point, the handle must be closed by calling `QueClosePoint()`. After calling `QueClosePoint()` your application must set the handle to `QueInvalidPointHandle`.
- To store a point between invocations of your application, you must store the serialized data using `QueSerializePoint()` before your application exits and re-create the point from the serialized data using `QueDeserializePoint()` when your application starts. You must never store a handle between invocations of your application.

## Making the Que API Library Available

In order for the Que API library to be accessible, the Que API Library patch must be installed on the handheld. The patch may be downloaded from the iQue 3600 Software Update Collection page of the Garmin web site. One way to navigate to this page is to select Software Updates from the Quick Links along the left side of the home page, then select iQue 3600 from the list of units.

## Opening and Closing the Que API Library

To get access to the Que API Library, `#include QueAPI.h` in your application.

Before the Que API Library can be used, it must be found or loaded, using the standard Palm OS® paradigm:

```
/*-----  
Find the Que API library. If not found, load  
it.  
-----*/  
error = SysLibFind( QueAPILibName,  
    &gQueAPILibRef );  
  
if ( error != errNone )  
{  
    error = SysLibLoad
```

```
(
    QueAPILibType,
    QueAPILibCreator,
    &gQueAPILibRef
);
ErrFatalDisplayIf( (error != errNone),
    "can't load QueAPI Library" );
}
```

Once the Que API Library is found and loaded, it must be opened by calling `QueAPIOpen()`. The `queAPIVersion` constant from `QueAPI.h` is supplied as a parameter to allow the library to determine if the version of the library expected by calling application is compatible with the version of the library that is loaded. `QueAPIOpen()` returns `queErrInvalidVersion` when the versions are not compatible:

```
error = QueAPIOpen( gQueAPILibRef,
    queAPIVersion );
ErrFatalDisplayIf( ( error ==
    queErrInvalidVersion ), "Incompatible
    version of QueAPILib." );
```

Once your application is done using the Que API Library (normally when the application stops), you should close and unload the library using the standard Palm OS® paradigm:

```
/*-----
Close the library.
-----*/
error = QueAPIClose( gQueAPILibRef );

/*-----
Unload the Library.
-----*/
if ( error == QueErrNone )
{
    SysLibRemove( gQueAPILibRef );
}
```

## Point Data Returned Through a Launch Code

Certain Que API Library APIs terminate the calling application in order to launch other applications to perform the work. When the work is done, the application specified in the API call is launched with a launch code that contains the resulting point data. This requires the application which receives the launch code to process this launch code data.

First, the `PilotMain()` procedure must handle the `sysAppLaunchCmdGoTo` launch code. The exact way this is handled depends on your application; however it will generally be handled identically to a `sysAppLaunchCmdNormalLaunch` with the addition of sending the goto data to the initial form before entering the event loop. If your application is already the current application, all that must be done is to send the goto data to the active form. This is illustrated below:

```
case sysAppLaunchCmdGoTo:
    /*-----
    If we have just been launched.
    -----*/
    if ( aLaunchFlags
        & sysAppLaunchFlagNewGlobals
        )
    {
        /*-----
        Start the application and go to
        the main form.
        -----*/
        AppStart();
        FrmGotoForm( MainForm );

        /*-----
        Send the goto data to the main
        form.
        -----*/
        HandleGoTo
            ( ( GoToParamsPtr ) aCmdPBP
            , MainForm
            );

        /*-----
        Enter the event loop and stop the
        application when done.
        -----*/
        AppEventLoop();
        AppStop();
    }

    /*-----
    Otherwise this is already the current
    application, just send the goto data
    to the main form.
```



```
-----*/  
else  
{  
    HandleGoTo  
        ( ( GoToParamsPtr ) aCmdPBP  
          , MainForm  
          );  
}  
break;
```

Second, the `HandleGoTo()` procedure must take the goto data from the launch command and send it to the form as a `frmGotoEvent` as shown below:

```
static void HandleGoTo  
( GoToParamsPtr aGoToParams  
  , const UInt16 aFormID  
  )  
{  
    EventType    event;  
  
    MemSet( &event, sizeof( EventType ), 0 );  
    event.eType = frmGotoEvent;  
    event.data.frmGoto.formID = aFormID;  
    event.data.frmGoto.recordNum =  
        aGoToParams->recordNum;  
    event.data.frmGoto.matchPos =  
        aGoToParams->matchPos;  
    event.data.frmGoto.matchLen =  
        aGoToParams->matchCustom;  
    event.data.frmGoto.matchFieldNum =  
        aGoToParams->matchFieldNum;  
    event.data.frmGoto.matchCustom =  
        aGoToParams->matchCustom;  
    EvtAddEventToQueue( &event );  
}
```

Third, the event handler for the form that will receive the `frmGotoEvent` must call `QueHandleEvent()`. If `QueHandleEvent()` returns true, `QueCreatePointFromEvent()` must be called to create a point from the event as shown below:

```
if ( QueHandleEvent  
     ( gQueAPILibRef  
     , aEventP )
```

```
)  
{  
    QueCreatePointFromEvent  
        ( gQueAPILibRef  
          , aEventP  
          , &gPoint  
        );  
}
```

The Que API Library APIs which return data through a launch code are:

- `QueCreatePointFromAddress()`
- `QueSelectAddressFromFind()`
- `QueSelectPointFromFind()`
- `QueSelectPointFromMap()`

See the Que API library example application in the SDK for a complete example of processing point data returned through a launch code.

## **The Que API Library and the Palm OS® Simulator**

The Garmin Palm OS® Simulator supports accessing map data. To make map data available to the simulator, a Palm database file containing the detailed map data, named `GMAPSUPP.PDB`, must be placed in the `AutoLoad` folder of the `GarminSimulator` folder. A Palm database file containing basemap data, named `GMAPBMAP.PDB`, may also be placed in the `AutoLoad` folder. Note that the combined size of the map files plus the other files in the `AutoLoad` folder cannot exceed the RAM size chosen in the simulator settings.

To create a `GMAPSUPP.PDB`:

- Follow the Map Install tool steps to select map sections for installation. The Map Install tool is available along the left side of the Garmin Palm Desktop.
- In the Device Setting section, select one of the iQue devices listed, then select Internal Storage for the Map Storage Location.
- After clicking the OK button on the “Transfer Complete – HotSync Required” dialog box, instead of performing a HotSync, move the `GMAPSUPP.PDB` from the Install folder for the device to the `AutoLoad` folder of the `GarminSimulator` folder.

To create a `GMAPBMAP.PDB`:

- Copy GMAPBMAP.PDB from the Basemap\AMR\_LITE folder of the Garmin install CD to the AutoLoad folder of the GarminSimulator folder.

## Que API Library Data Structures

### Basic Data Types

uint8	Unsigned 8 bit integer.
uint16	Unsigned 16 bit integer.
uint32	Unsigned 32 bit integer.
sint8	Signed 8 bit integer.
sint16	Signed 16 bit integer.
sint32	Signed 32 bit integer.
TCHAR	Char type.

### QuePositionDataType

QuePositionDataType specifies the 3 dimensional position of a point.

```
typedef struct
{
    sint32    lat;
    sint32    lon;
    float     altMSL;
} QuePositionDataType;
```

### Field Descriptions

lat	The latitude of the point in semicircles. Semicircles are described in GPS data structure GPSPositionDataType.
lon	The longitude of the point in semicircles. Semicircles are described in GPS data structure GPSPositionDataType.
altMSL	The altitude above mean sea level of the point in meters. This field is not used.

## QuePointType

QuePointType specifies the information about the position that is available to an application.

```
typedef struct
{
    char                id[ quePointIdLen ];
    QueSymbolT16        smbl;
    QuePositionDataType posn;
} QuePointType;
```

### Field Descriptions

id	A NULL-terminated string containing the name of the point.
smbl	The symbol associated with the point. This field is not used.
posn	The 3 dimensional position of the point.

## QueSelectAddressType

QueSelectAddressType specifies the address fields that can be supplied when creating a point at an address.

```
typedef struct
{
    const TCHAR *streetAddress;
    const TCHAR *city;
    const TCHAR *state;
    const TCHAR *country;
    const TCHAR *postalCode;
} QueSelectAddressType;
```

### Field Descriptions

streetAddress	A pointer to a NULL-terminated string containing the street number and street name of the address.
---------------	--

<code>city</code>	A pointer to a NULL-terminated string containing the city of the address.
<code>state</code>	A pointer to a NULL-terminated string containing the state of the address.
<code>country</code>	A pointer to a NULL-terminated string containing the country of the address.
<code>postalCode</code>	A pointer to a NULL-terminated string containing the postal code of the address.

## Que API Library Constants

### Error Codes

<code>queErrNone</code>	Success.
<code>queErrNotOpen</code>	attempted to close the library without opening it first.
<code>queErrBadArg</code>	Invalid parameter passed.
<code>queErrMemory</code>	Out of memory.
<code>queErrNoData</code>	No data available.
<code>queErrAlreadyOpen</code>	The library is already open.
<code>queErrInvalidVersion</code>	The library is an incompatible version.
<code>queErrCmndUnavail</code>	The command is unavailable.
<code>queErrStillOpen</code>	Library is still open.
<code>queErrFail</code>	General failure.
<code>queErrCancel</code>	Action cancelled by user.

### Other values

<code>quePointIdLen</code>	Length of the point identifier string including the NULL-termination character.
<code>queInvalidSemicircles</code>	Invalid semicircle value.
<code>queInvalidAltitude</code>	Invalid altitude value.
<code>queInvalidPointHandle</code>	Invalid point handle.
<code>queInvalidSymbol</code>	Invalid symbol value.

## Que API Library Functions

### QueAPIClose

<b>Purpose</b>	Closes the Que API Library.	
<b>Prototype</b>	<code>QueErrT16 QueAPIClose( const UInt16 refNum )</code>	
<b>Parameters</b>	<code>-&gt; refNum</code>	Reference number for the library.
<b>Result</b>	<code>queErrNone</code>	No error.
	<code>queErrNotOpen</code>	The library is not open.
	<code>queErrStillOpen</code>	Couldn't be closed because the library is still in use by other applications.
<b>Comments</b>	Closes the Que API Library and disposes of the global data memory if required. Called by any application or library that's been using the Que API Library and is now finished with it.  This should not be called if <code>QueAPIOpen</code> failed.  If <code>queErrStillOpen</code> is returned, the calling app should not call <code>SysLibRemove()</code> .	

### QueAPIOpen

<b>Purpose</b>	Opens the Que API Library.	
<b>Prototype</b>	<code>QueErrT16 QueAPIOpen( const UInt16 refNum, const UInt16 version )</code>	

<b>Parameters</b>	-> refNum	Reference number for the library.
	-> version	Version of library expected by the application.
<b>Result</b>	queErrNone	No error.
	queErrMemory	Unable to get memory for the library.
	queErrInvalidVersion	The expected version of the library is not compatible with this library.
<b>Comments</b>	<p>Opens the Que API Library and prepares it for use. Called by any application or library that wants to use the services that the library provides.</p> <p>QueAPIOpen ( ) must be called before calling any other Que API Library functions. If the return value is anything other than queErrNone the library was not opened.</p>	

## QueClosePoint

<b>Purpose</b>	Closes the handle to a point.	
<b>Prototype</b>	<pre>QueErrT16 QueClosePoint(     const UInt16 refNum,     const QuePointHandle point )</pre>	
<b>Parameters</b>	-> refNum	Reference number for the library.
	-> point	Point handle to be closed.
<b>Result</b>	queErrNone	No error.
	queErrBadArg	The point handle was not open.
<b>Comments</b>	<p>Closes the handle to a point. This must be called for all open point handles before exiting your application. After calling this procedure, the caller should set the point handle to queInvalidPointHandle to indicate that it has been closed.</p>	

## QueCreatePoint

<b>Purpose</b>	Creates a point with the specified data.
----------------	--

## Que API Library

### *Que API Library Functions*

---

<b>Prototype</b>	<pre>QueErrT16 QueCreatePoint( const UInt16 refNum, const QuePointType *pointData, QuePointHandle *point )</pre>	
<b>Parameters</b>	-> refNum	Reference number for the library.
	-> pointData	Pointer to the data to use when creating the point.
	<- point	Contains the point handle of the created point.
<b>Result</b>	queErrNone	No error.
	queErrMemory	Unable to get memory for the point.
	queErrBadArg	The point handle was already open.
<b>Comments</b>	If an error occurs, the returned point handle may not be open.	

## QueCreatePointFromAddress

**Purpose** Creates a point from the specified address data.

**Prototype**

```
QueErrT16 QueCreatePointFromAddress(  
const UInt16 refNum,  
const QueSelectAddressType *address,  
const UInt32 relaunchAppCreator )
```

<b>Parameters</b>	-> refNum	Reference number for the library.
	-> address	Pointer to the address data to use when creating the point.
	-> relaunchAppCreator	Creator ID of the application to launch when the point has been created.
<b>Result</b>	queErrNone	No error.
	queErrMemory	Unable to get the library's global data.
<b>Comments</b>	Creates a point at the location of the specified address.	



---

**IMPORTANT:** This call terminates the calling application and returns the results through a launch code. See the **Point Data Returned Through a Launch Code** section for more details.

---

Not all fields of the input address data need to be supplied; a match will be attempted using the fields that contain data. Any unused fields should be set to NULL.

If a single address match cannot be found an invalid point handle will be returned through the launch code.

## QueCreatePointFromEvent

**Purpose** Creates a point from a frmGotoEvent that contains point data.

**Prototype** `QueErrT16 QueCreatePointFromEvent(  
const UInt16refNum, const EventType*event,  
QuePointHandle *point )`

<b>Parameters</b>	-> refNum	Reference number for the library.
	-> event	Pointer to the frmGotoEvent that contains point data.
	<- point	Contains the point handle of the created point.

<b>Result</b>	queErrNone	No error.
	queErrNoData	The event did not contain the necessary data.
	queErrBadArg	The point handle was already open.
	queErrMemory	Unable to allocate memory for the point or unable to get the library's global data.

**Comments** Creates a point from the frmGotoEvent data. This event is sent when the specified application is launched after calling certain Que API library APIs. See the **Point Data Returned Through a Launch Code** section for more details. This should only be called if the result of QueIsFindResultEvent( ) is true.

If an error is returned the point handle will not be open.

## QueDeserializePoint

**Purpose** Creates a point from serialized point data.

**Prototype**

```
QueErrT16 QueDeserializePoint(  
    const UInt16 refNum, const void *pointData,  
    const UInt32 pointDataSize,  
    QuePointHandle *point )
```

<b>Parameters</b>	-> refNum	Reference number for the library.
	-> pointData	Pointer to the serialized point data.
	-> pointDataSize	Size in bytes of the serialized point data.
	<- point	Contains the point handle of the created point.

<b>Result</b>	queErrNone	No error.
	queErrBadArg	The point handle was already open, the pointer to the serialized data was NULL, the point data size was incorrect, or the format of the serialized point data was not recognized.
	queErrMemory	Unable to allocate memory for the point or unable to get the library's global data.

**Comments** Creates a point from the serialized point data created by QueSerializePoint(). See the description of QueSerializePoint() for more information.

If an error is returned the point handle will not be open.

## QueGetPointInfo

**Purpose** Returns information about the point.

**Prototype**

```
QueErrT16 QueGetPointInfo( const UInt16 refNum,  
    const QuePointHandle point,  
    QuePointType *pointInfo )
```

<b>Parameters</b>	-> refNum	Reference number for the library.
	-> point	Point handle from which to get information.
	<- pointInfo	Contains the information about the point.
<b>Result</b>	queErrNone	No error.
	queErrBadArg	The point handle was not open.

## QueHandleEvent

**Purpose** Handles Que API library events.

**Prototype** `Boolean QueHandleEvent( const UInt16 refNum,  
const EventType *event )`

<b>Parameters</b>	-> refNum	Reference number for the library.
	-> event	Pointer to the event.

**Result** Returns true if the event contains contains point data. If the event contains point data then `QueCreatePointFromEvent( )` can be called to create a point from the event.

**Comments** This should be called in the active form's event loop because there is other Que API library processing performed during this call.

## QueRouteToPoint

**Purpose** Creates a route from the current location to the point.

**Prototype** `QueErrT16 QueRouteToPoint( const UInt16 refNum,  
const QuePointHandle point,  
const Boolean showMap )`

<b>Parameters</b>	-> refNum	Reference number for the library.
	-> point	Point handle to route to.
	-> showMap	true to terminate calling application and activate the QueMap application centered on the vehicle, false to remain in the calling application.

<b>Result</b>	queErrNone	No error.
	queErrBadArg	The point handle is not open.
	queErrMemory	Unable to get the library's global data.

---

**IMPORTANT:** If `showMap` is true, this call terminates the calling application.

---

## QueSelectAddressFromFind

**Purpose** Allows the user to create a point by selecting an address from the find address form.

**Prototype**

```
QueErrT16 QueSelectAddressFromFind(
    const UInt16 refNum,
    const QueSelectAddressType *address,
    const UInt32 relaunchAppCreator,
    const Boolean tryToCreateFirst )
```

<b>Parameters</b>	-> refNum	Reference number for the library.
	-> address	Pointer to the address data to use when selecting the point.
	-> relaunchAppCreator	Creator ID of the application to launch when the point has been created.
	-> tryToCreateFirst	Tries to create the point from the address data before displaying the find address form. See comments for more details.

<b>Result</b>	queErrNone	No error.
	queErrMemory	Unable to get the library's global data.

**Comments** Displays the QueFind address form to allow the user to select an address from which to create a point.

---

**IMPORTANT:** This call terminates the calling application and returns the results through a launch code. See the **Point Data Returned Through a Launch Code** section for more details.

---

The fields of the address form will be pre-filled with the supplied address data. Not all fields of the input address data need to be supplied; any unused fields should be set to NULL.

If `tryToCreateFirst` is true, this call will first attempt to create a point at the location of the specified address exactly like `QueCreatePointFromAddress()`. If a single address match is found, it will be returned through the launch code and the `QueFind` address form will not be displayed. If a single address match cannot be found, then the `QueFind` address form is displayed exactly as if this was called with `tryToCreateFirst` set to false.

If the user cancels finding an address an invalid point handle will be returned through the launch code.

## QueSelectPointFromFind

**Purpose** Allows the user to create a point by selecting an item using `QueFind`.

**Prototype** `QueErrT16 QueSelectPointFromFind(  
const UInt16 refNum,  
const UInt32 relaunchAppCreator )`

<b>Parameters</b>	<code>-&gt; refNum</code>	Reference number for the library.
	<code>-&gt; relaunchAppCreator</code>	Creator ID of the application to launch when the point has been created.

<b>Result</b>	<code>queErrNone</code>	No error.
	<code>queErrMemory</code>	Unable to get the library's global data.

**Comments** Displays `QueFind` to allow the user to select an item from which to create a point. This is similar to `QueSelectAddressFromFind()` except this will display the top-level `QueFind` page.

---

**IMPORTANT:** This call terminates the calling application and returns the results through a launch code. See the **Point Data Returned Through a Launch Code** section for more details.

---

If the user cancels finding an item an invalid point handle will be returned through the launch code.

## QueSelectPointFromMap

<b>Purpose</b>	Allows the user to create a point by selecting it from a map.	
<b>Prototype</b>	<pre>QueErrT16 QueSelectPointFromMap(     const UInt16 refNum,     const UInt32 relaunchAppCreator )</pre>	
<b>Parameters</b>	-> refNum	Reference number for the library.
	-> relaunchAppCreator	Creator ID of the application to launch when the point has been created.
<b>Result</b>	queErrNone	No error.
	queErrMemory	Unable to get the library's global data.
<b>Comments</b>	Allow the user to create a point by tapping a location on a displayed map.	

---

**IMPORTANT:** This call terminates the calling application and returns the results through a launch code. See the **Point Data Returned Through a Launch Code** section for more details.

---

If the user cancels the operation an invalid point handle will be returned through the launch code.

## QueSerializePoint

<b>Purpose</b>	Returns the serialized data that represents the point.	
<b>Prototype</b>	<pre>UInt32 QueSerializePoint( const UInt16 refNum,     const QuePointHandle point,     void *pointData, const UInt32 pointDataSize )</pre>	
<b>Parameters</b>	-> refNum	Reference number for the library.
	-> point	Point handle to serialize.
	<- pointData	Contains the serialized data.
	-> pointDataSize	Size in bytes of the pointData buffer.

**Result** Returns the size in bytes of the serialized data.

**Comments** Returns the serialized data (i.e. series of bytes) that represents the point. This is used for long-term storage of the point. The point can be re-created by calling `QueDeserializePoint()`.

This always returns the size in bytes of the serialized data. If the supplied buffer is not large enough to hold all the serialized data, no data will be written into the buffer.

Typical usage is to call `QueSerializePoint()` once with `pointData` set to `NULL` and `pointDataSize` set to 0, then use the returned size to allocate a buffer to hold the serialized data. Then call `QueSerializePoint()` again with the address and size of the allocated buffer.

---

**IMPORTANT:** Never set `pointData` to `NULL` without setting `pointDataSize` equal to 0.

---

## QueSetRouteToItem

**Purpose** Sets the route form “Route to” item to the point.

**Prototype**

```
QueErrT16 QueSetRouteToItem(  
    const UInt16 refNum,  
    const QuePointHandle point )
```

**Parameters**

-> refNum	Reference number for the library.
-> point	Point handle to set the “Route to” item to, or an invalid point handle to clear the “Route to” item.

**Result**

queErrNone	No error.
queErrMemory	Unable to get the library’s global data.
queErrFail	Unable to set the “Route to” item.

**Comments** Sets the “Route to” item on the route form to the specified point. The “Route to” item will be cleared when the library is closed, which normally will happen when the application using the library exits. The

item can be cleared by calling `QueSetRouteToItem( )` with a point handle value of `queInvalidPointHandle`.

The “Route to” item is the context-sensitive item that is displayed above the route icons when `QueRoutes` is displayed modally by tapping the route icon in the graffiti area or in the status bar.

## **QueViewPointDetails**

**Purpose** Displays a modal form containing a map and other details about the point.

**Prototype** `QueErrT16 QueViewPointDetails(  
const UInt16 refNum,  
const QuePointHandle point )`

**Parameters**

-> refNum	Reference number for the library.
-> point	Point handle to view the details of.

**Result**

<code>queErrNone</code>	No error.
<code>queErrMemory</code>	Unable to get the library’s global data.
<code>queErrBadArg</code>	The point handle is not open.

## **QueViewPointOnMap**

**Purpose** Switches to the `QueMap` application centered on the point.

**Prototype** `QueErrT16 QueViewPointOnMap(  
const UInt16 refNum,  
const QuePointHandle point )`

**Parameters**

-> refNum	Reference number for the library.
-> point	Point handle to view on map.

**Result**

<code>queErrNone</code>	No error.
<code>queErrMemory</code>	Unable to get the library’s global data.
<code>queErrBadArg</code>	The point handle is not open.



**Comments** This terminates the calling application and launches the QueMap application centered on the specified point.

---

**IMPORTANT:** This call terminates the calling application.

---