



POWERED BY
gameSPY®

Powered By GameSpy Help File

Thank you for choosing GameSpy's online middleware for use in your online game. GameSpy's software development kits (SDKs), combined with GameSpy's reliable backend services, will help provide your players with a rewarding online experience.

This help file provides you with detailed documentation on every SDK in the Powered By GameSpy product line. In addition to the reference documentation in this file you will also find answered to common questions in [the GameSpy Support Knowledge Base](#).

If you are unable to find the answer to your question in the documentation or knowledge base you may contact developer support by [e-mailing devsupport@gamespy.com](mailto:devsupport@gamespy.com). Please note, you must be a licensed developer (have a development or publisher's deployment license) with GameSpy to be eligible for developer support.

The latest version of this help file is available from our [secure developer site](#) as a part of the Common Code download.

Available Services Check

Overview

The GameSpy SDKs provide a way to check if a game's backend services are available. Any application that uses these services must first check if they are available before using the actual SDKs. The check does not need to be made before using SDKs which do not communicate with the GameSpy backend (currently Chat, GHTTP, GT2, and Voice). If you attempt to use an SDK that does communicate with the backend before checking that the backend is currently available, the SDK will fail to initialize.

The file `available.h`, included in the `goacommon.zip`, has the necessary function prototypes for making the availability check. `available.c` is the source file, and should generally be compiled directly into your project (along with the rest of the GameSpy code being used). To start the availability check simply call `GSIStartAvailableCheck()`, passing it your gamename. This will initiate a request with the backend to see if your game's backend services are available. After initiating the request, `GSIAvailableCheckThink()` should be called to let the code process incoming replies and send retries. It should be continued to be called as long as it returns `GSIACWsaiting`. It is not very time-sensitive, so it does not need to be called more frequently than every 100ms (although it can be). If the check needs to be aborted for any reason, for example due to the player leaving the online area of the game, `GSICancelAvailableCheck()` should be called to do any needed cleanup.

As soon as `GSIAvailableCheckThink()` returns any value other than `GSIACWaiting`, the check has completed. No extra cleanup is needed. The return value indicates the result of the check. If it is `GSIACAvailable` then the game's backend services are available, and the game can continue to use the GameSpy SDKs normally. If the return value is `GSIACUnavailable` or `GSIACTemporarilyUnavailable`, then the game's backend services are not available, and the game should not use any GameSpy SDKs that rely on backend services. If the user attempts to use online aspects of the game that are not available, the game should show appropriate messaging to the user. If the return value was

GSIACUnavailable, then the game should inform the user that the game's online component is no longer supported. If the return value was GSIACTemporarilyUnavailable, then the game should inform the user that the game's online component is currently unavailable and they should try again later.

GSIACAvailable will also be returned from GSIACAvailableCheckThink() if no response is received from the request initiated by GSIACStartAvailableCheck(). In other words, if it cannot be determined if the backend is available, the safe assumption that it is available is made. The case of a failed initialization or connection for each of the individual SDKs should always be handled.

GSIACAvailableCheckThink Return Values:

- **GSIACWaiting** Continue to call GSIACAvailableCheckThink. Processing has not yet completed.
- **GSIACAvailable**
This game's backend services are available. Continue normal operations.
- **GSIACUnavailable**
This game's backend services are not available. Game play will not be possible for an extended length of time or indefinitely. This should only occur when a service has been discontinued because the developer or publisher has chosen to not renew the service.

PC games will continue to be supported in GameSpy's Arcade.

- **GSIACTemporarilyUnavailable**
This game's backend services are temporarily unavailable. Reserved for scheduled downtime. Services should be restored momentarily.

Game UI

The SDK does not make any assumptions or requirements as to what messages should be displayed to the users. Developers are free to implement whatever appropriate messages they wish. What the message should say will be dependent on several factors including the return value and if the developer or publisher has opted to not take the co-branding discounts.

We recommend that developers implement generalized messages, for example:

If `GSIAvailableCheckThink` returns with as `Unavailable` a suitable messages might be:

- "Online support for Tony Hawk: Underground is no longer available."
- "Midwaysports.net is no longer available for Blitz 2010."

If `GSIAvailableCheckThink` returns with as `TemporarilyUnavailable` a suitable message might be:

- "Online play for Hidden and Dangerous is temporarily unavailable do to maintenance."
- "Midwaysposrts.net is temporarily unavailable. Please try again soon."

Testing

Two special gamenames are reserved for testing client-side availability check code. Calling `GSISStartAvailableCheck()` with a gamename of "unavailable" will cause the availability check to return `GSIACTemporarilyUnavailable`. Using a gamename of "tempunavail" will cause the availability check to return `GSIACTemporarilyUnavailable`.

Requirements

Performing the check is a TRC requirement as of November 10th, 2003. All titles will be tested to be sure they are implementing the check correctly.

Overview

As consumer product companies continue to aggressively pursue opportunities to market and promote their products to the 18-34 year old males, a demographic that has abandoned television and other traditional advertising outlets in favor of playing computer and video games, game publishers are well-positioned to take advantage of this opportunity through product placement and advertising integration into games.

The Marketing SDK provides the tools and services necessary to make this possible by allowing the game publisher to dynamically serve product placements and advertisements into games, modify and change them as desired, and track and measure the usage and performance of those advertisements.

The traditional scenario of hard-coding the advertising assets into a game forced the publisher to have signed agreements in place, artwork created and integrated into the game, and all the requisite approvals in place before the game reaches beta. Using the Marketing SDK, publishers can provision product placements and advertisements when they and their partners are ready for them.

[\(back to top\)](#)

Project Setup

Files to include

The Marketing SDK leverages the gHTTP and gSOAP libraries to provide a mature and robust network transport layer. In addition, the GameSpy common code is used to provide standardized data typing across supported platforms. These libraries must be included in the project.

Common Code

The source files found in the root SDK directory must be included in the project. The common code contains platform specific type definitions and shared utility functions.

gHTTP

The GameSpy HTTP SDK is used when downloading files to disk or when streaming data into memory. All files within the /ghttp folder should be included. (Samples and subdirectories should be omitted.)

gSOAP

This commercial SOAP library is used when querying for the active ad units and when reporting usage statistics. More information about gSOAP may be found on the gSOAP website at <http://www.cs.fsu.edu/~engelen/soap.html>

The gSOAP files may be found in the “/gsoap” directory.

Marketing SDK Files

All of the files within the “/Ad” directory must be included in the project. Files within the “/Ad/AdSoap” directory must also be included.

[\(back to top\)](#)

Integrating the SDK

Compile-time options

A variety of settings are defined at the top of Ad.h to control memory and bandwidth usage. These optional settings are fully detailed in the reference section of this document.

PS2 developers must define WITH_LEAN and WITH_LEANER. This reduces the size of the gSOAP library and removes code that may not be compatible with all network stacks.

[\(back to top\)](#)

Initialize the SDK

The Marketing SDK must be initialized before it may be used. It is recommend that the SDK be initialized in the following manner:

```
AdInterfacePtr anInterface = NULL;
AdResult       aResult     = AdResult_NO_ER
AdInitParams   anInitParams;

memset(&anInitParams, 0, sizeof(AdInitParams
anInitParams.mGameId = GAME_ID;

aResult = adInitialize(&anInitParams, &anInt
if (aResult != AdResult_NO_ERROR)
    printf("adInitialize failed (%d)\r\r
```

The AdInitParams structure contains runtime settings that may be used to control SDK behavior.

```
typedef struct
{
    gsi_i32      mGameId;
    const char* mQueryHostOverrideURL; // Override t

    // For offline usage stats
    gsi_bool     mOfflineOnly;         // (e.g. sing
    const char*  mOfflineFilePath;    // relative t

    // For ad download caching
    const char*  mCachePath;          // cache dire

} AdInitParams;
```

[\(back to top\)](#)

Register each ad position

At the beginning of the game (or at the start of each level) the SDK must be told which ad positions are in use. In addition to the position name, information about a default advertisement must be supplied. The default advertisement will be used if network conditions prevent an ad download or if an ad file is corrupted.

```
AdResult AD_CALL adRegisterPosition(const AdInterface  
                                     cons  
                                     AdUr  
                                     cons  
                                     cons
```

Although a default ad is generally unbranded, the SDK will continue to collect usage statistics that will be visible through the publisher's portal. Therefore, it is important that the default ad have a valid AdUnitID.

[\(back to top\)](#)

Query the active ad for each position

The logic for selecting advertisements and matching user information for targeted delivery is contained within the AdServer. The SDK must simply pass up the user information along with a list of ad positions and the ad server will return a list of advertisements to fill those positions.

Currently, the SDK supports targeting based on birthdate only. The user's profileid is used to count unique downloads.

```
AdResult AD_CALL adQueryForActiveUnits(  
    AdInterfacePtr theInterface,  
    gsi_u32 theProfileI  
    gsi_u32 theBirthDat  
    AdQueryForActiveUnit  
    gsi_time theTimeoutM
```

This is an asynchronous query, so a callback and timeout parameter are provided.

[\(back to top\)](#)

Download new creatives

The SDK can begin downloading new creatives as soon as the query for active ads has completed.

```
AdResult AD_CALL adDownloadNewCreatives(  
    AdInterfacePtr theInterface,  
    gsi_i32         theThrottle,  
    AdDownloadNewCreativesProgressCallback thePr  
    AdDownloadNewCreativesCompletedCallback theC  
    gsi_time       theTimeoutMs);
```

Win32 and Mac developers may take advantage of ad caching when using ad rotations. Caching is performed automatically by the SDK. When the `adDownloadNewCreatives` function is called, the SDK will check if the required ad exists within the local cache. If the file is found, the cached file will be used in place of a new download. CRC checks are performed to ensure ad integrity.

When developing on the PS2, the SDK will not store files to the memory card. Instead, the developer must process data received in `theProgressCallback` and copy it to the desired memory location.

When setting up advertisements in the publisher portal there are some cases where you may want to provide a URL for content, but do not want the SDK to auto-download it. I recommend prefixing the URL with a token to identify the type of content. The presence of this token will invalidate the URL causing the SDK to ignore it.

For example, if the movie url is “`http://localhost/movie.swf`” you might use “`stream:http://localhost/movie.swf`”. The game client can then detect the presence of the “`stream`” token and begin streaming the movie. Note that a crc value is not required when using this method since no download is being performed by the SDK.

[\(back to top\)](#)

Notify the SDK when ads are used

An ad download is usually binary data and may contain multiple game resources. The SDK has no knowledge of the internal file contents, so it's left up to the developer to inform the SDK when an ad is on screen or is being interacted with.

There are two pre-defined categories for ad usage. The interpretation of the category names is somewhat arbitrary, but here are some helpful guidelines for when each category should be used.

UC_VIEWS

Usually defined as “time on screen”. Viewing a billboard, floating blimp or other passive impressions would fall into this category. e.g. The branded item exists for aesthetic purposes only, is not used in gameplay.

UC_INTERACTIONS

Drinking a branded soda or constructing a branded storefront would fall into this category. Please note that viewing a branded soda would fall into the UC_VIEWS category. Developers are free to use whichever category they prefer, but since this affects usage reporting we recommend that a clear separation is chosen.

```
AdResult AD_CALL adBeginTrackUsageTime(AdInterfacePt
    const char*     thePositionName,
        AdUsageCategory theCat);
AdResult AD_CALL adEndTrackUsageTime  (AdInterfacePt
                                         C
                                         A
AdResult AD_CALL adIncrementUsageCount(AdInterfacePt
                                         C
                                         A
```

As an example, imagine that you have a branded blimp that will fly through the users field of view. When the blimp appears on screen you should call `adBeginTrackUsageTime`. When the blimp explodes (or peacefully floats off-screen) you should call `adEndTrackUsageTime`.

You must call `adEndTrackUsageTime` once for each call to `adBeginTrackUsageTime`. This allows for simple tracking when multiple blimps are on screen. If you call `adBeginTrackUsageTime` five times, but call `adEndTrackUsageTime` only four times, the ad will still be considered "in use".

Calling `adBeginTrackUsageTime` multiple times will not inflate the viewing time. If three blimps on are screen for five seconds, you are credited for five seconds of viewing time. (not 15)

[\(back to top\)](#)

Report usage statistics

We recommend that the `adSendUnitUsageData` function be called at least once every five minutes throughout the game session, and once again when the game session ends. This is a flexible guideline and may vary depending on game type.

[\(back to top\)](#)

Ad Metrics

Developers who are familiar with web based metrics may recall the terms “impression” and “click-through”. These are somewhat restricted usage metrics which are fit to current web server technology.

GameSpy client side ad reporting is much more robust and will increase the value of your ad inventory.

Views (UC_VIEWS)

This is usually interpreted as when an ad is on-screen.

Web developers may notice a similarity with ad “impressions”, however impression count is a limited report metric. GameSpy supports time based measurements which are unavailable in a web environment.

For example, when playing a flash movie in-game, the GameSpy Marketing SDK is able to report not only the number of times the movie was streamed, but also how many seconds the movie was viewed. This is not as simple as multiplying the number of downloads by the length of the movie. When given the option, your games gamers may close the advertisement before it has finished playing, or they may watch an interesting movie 3 or 4 times!

Interactions (UC_INTERACTIONS)

The interactions category is left for general developer use. Actions such as picking up a health pack or firing a gun are suitable for interactions.

[\(back to top\)](#)

Reference

Compile Time Options

These are defined at the top of ad.h.

GSI_AD_STATIC_MEM

When defined, this will cause the SDK to prefer static memory over dynamic memory. Array sizes must be defined at compile time and array growth will not be permitted.

GSI_AD_DEFAULT_FILE_NAME (“_gsiad.dat”)

File name used to store offline stats.

GSI_AD_MAX_TRANSFER_COUNT (1)

Specifies the number of simultaneous downloads the SDK will perform. The default is set to single downloads. Increasing this may result in faster download speeds, but will require additional memory for the ghttp sdk.

GSI_AD_MAX_FILENAME_LENGTH (255)

Buffer size for filenames. Reduce this only if you are extremely strapped for memory.

GSI_AD_POSITION_COUNT (5)

Maximum number of positions in any one level/map. This is the number of positions the SDK will keep in memory.

GSI_AD_MAX_POSITION_NAME_LENGTH (32)

Position names are string identifiers for ad positions.

GSI_AD_UNIT_ARRAY_INITIAL_CAPACITY (10)

Starting capacity for the unit array. You will usually need 2 x number of positions.

GSI_AD_UNIT_ARRAY_MAX_CAPACITY (15)

The maximum size that the unit array will grow to.

GSI_AD_UNIT_ARRAY_GROWBY (0)

Array growth size. The array will grow as needed until the max capacity is reached. The default of zero prevents memory growth and is required when using *GSI_AD_STATIC_MEM*.

GSI_AD_MAX_EXTRA_DATA_LENGTH (64)

Maximum buffer size of developer data. This data is associated with the ad using the publisher portal. Increase this if you require additional data.

GSI_AD_MAX_URL_LENGTH (255)

Maximum size of download URLs specified using the ad portal. Increase this if you require support for longer URLs.

[\(back to top\)](#)

Multithreading

The gSOAP library is a blocking TCP socket library, this requires that we run it in a dedicated thread.

The GameSpy common code will automatically create and destroy threads for gSOAP as needed. One thread is required for each outstanding gSOAP call. The stack size required by this thread will vary by platform. 8k is usually large enough as long as WITH_LEAN and WITH_LEANER are defined.

Because the gSOAP library is only used when retrieving a list of ads or when reporting usage statistics, the library will not affect performance during gameplay.

[\(back to top\)](#)

FAQ

While using the SNSystems stack for the PS2 I receive network errors from the update query. What might be happening?

PS2 developers using SNSystems must specify the number of threads that have network access when they call sockAPIinit. This number must include an extra thread for the gsoap library.

How often should usage data be reported?

We recommend that adSendUnitUsageData be called at startup, shutdown and periodically throughout the game (~5 minutes.) If the player's network connection drops, usage data may be saved to an offline file. Sending usage data at program startup ensures that the offline data will be reported.

How can I create advertisements and set locations for my title?

Administrative functions may be found on the Ad Portal web site. For access to this site, please contact devsupport@gamespy.com. Once you have been granted access, please check the "Guide" section of the web site for helpful tips and directions.

Why do the cache files have a "bin" extension?

Advertisement creatives may contain any type of binary data, including zip files and self extracting executables. We save creatives as "bin" files to protect users against harmful file types, and use a check sum to verify the cache file integrity once it has been fully downloaded.

I can't use the cache files because my title requires the file extension. What do you recommend?

We recommend renaming or copying the cache file into a new folder. If the cache file is removed the SDK may download it again at a future time. To prevent this, the default ad for the position may be updated with the new file name.

How can I see/test an advertisement in game before making it active?

Since advertisements are position based, we recommend using special position names for test builds of the game. In the future we may consider supporting ads that would be delivered only to “test” clients. Please contact devsupport@gamespy.com if you are interested in this or other feature requests.

[\(back to top\)](#)

gSOAP License Notice

Part of the software embedded in this product is gSOAP software.

Portions created by gSOAP are Copyright (C) 2001-2004 Robert A. van Engelen, Genivia inc. All Rights Reserved.

THE SOFTWARE IN THIS PRODUCT WAS IN PART PROVIDED BY GENIVIA INC AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

[\(back to top\)](#)

Advertising SDK Functions

adBeginTrackUsageTime	Begins time-based usage tracking for the specified unit.
adCancelDownloads	Cancel any Creative downloads that are in progress.
adCancelQueryForActiveUnits	Cancels a pending query.
adDownloadNewCreatives	Begins downloading new creatives as necessary.
adEndTrackUsageTime	Ends time-based usage tracking for the specified unit.
adGetUnitInfoByID	Retrieves the AdUnitInfo the specified unit.
adGetUnitInfoByPosition	Retrieves the AdUnitInfo for the unit in the specified position.
adIncrementUsageCount	Increment ad usage (without tracking time in use)
adInitialize	Create an SDK instance.
adQueryForActiveUnits	Queries the list of active ad units from the ad server

adRegisterPosition	Registers the string name and default ad of a new position.
adReset	Return the SDK back to its initialization point.
adSendUnitUsageData	Uploads usage data to the Ad Server.
adShutdown	Destroys the SDK interface object and frees any allocated memory.
adThink	Allows the SDK to continue processing.

adBeginTrackUsageTime

Begins time-based usage tracking for the specified unit.

```
AdResult adBeginTrackUsageTime(  
    AdInterfacePtr theInterface,  
    const char * thePositionName,  
    AdUsageCategory theCat );
```

Routine	Required Header	Distribution
adBeginTrackUsageTime	<ad.h>	SDKZIP

Return Value

This function returns `AdResult_NO_ERROR` upon success. Otherwise a valid `AdResult` error condition is returned. (see remarks)

Parameters

theInterface

[in] SDK interface previously initialized using `adInitialize`

thePositionName

[in] String name of the position, registered with `adRegisterPosition`

theCat

[in] UC_VIEWS or UC_INTERACTIONS

Remarks

This function will mark the specified position as "in use" until `adEndTrackUsageTime` is called. Two separate metrics are provided, `UC_VIEWS` and `UC_INTERACTIONS`. These may be used to track arbitrarily different types of usage on the same object.

For example, a branded gun that is visible may be tracked using `UC_VIEWS`. A branded gun that is fired may be tracked using `UC_INTERACTIONS`.

Please note that a single Ad may appear in multiple positions. Multiple calls to **`adBeginTrackUsageTime`** will be ignored. In other words, the same image appearing in two different locations will result in one time measurement.

This function may return:

`AdResult_NO_ERROR`

`AdResult_INVALID_PARAMETERS`

`AdResult_POSITION_NOT_FOUND`

`AdResult_UNIT_NOT_FOUND`.

Section Reference: [Gamespy Advertising SDK](#)

See Also: [adEndTrackUsageTime](#), [adIncrementUsageCount](#)

adCancelDownloads

Cancel any Creative downloads that are in progress.

```
AdResult adCancelDownloads(  
    AdInterfacePtr theInterface );
```

Routine	Required Header	Distribution
adCancelDownloads	<ad.h>	SDKZIP

Return Value

This function returns `AdResult_NO_ERROR` upon success. Otherwise a valid `AdResult` error condition is returned. (see remarks)

Parameters

theInterface

[in] SDK interface previously initialized using `adInitialize`

Remarks

Multiple downloads may be in progress or pending. This call will cancel all of them. A progress callback will be triggered for each download that is canceled.

This function may return:
AdResult_NO_ERROR.

Section Reference: [Gamespy Advertising SDK](#)

See Also: [adDownloadNewCreatives](#)

adCancelQueryForActiveUnits

Cancels a pending query.

```
AdResult adCancelQueryForActiveUnits(  
    AdInterfacePtr theInterface );
```

Routine	Required Header	Distribution
adCancelQueryForActiveUnits	<ad.h>	SDKZIP

Return Value

This function returns `AdResult_NO_ERROR` upon success. Otherwise a valid `AdResult` error condition is returned. (see remarks)

Parameters

theInterface

[in] SDK interface previously initialized using `adInitialize`

Remarks

Cancels a previous call to `adQueryForActiveUnits`. A callback will be triggered with the status `AdResult_CANCELLED`.

This function may return:

`AdResult_NO_ERROR`

`AdResult_INVALID_PARAMETERS`.

Section Reference: [Gamespy Advertising SDK](#)

See Also: [adQueryForActiveUnits](#)

adDownloadNewCreatives

Begins downloading new creatives as necessary.

```
AdResult adDownloadNewCreatives(  
    AdInterfacePtr theInterface,  
    gsi_i32 theThrottle,  
    AdDownloadNewCreativesProgressCallback  
    theProgressCallback,  
    AdDownloadNewCreativesCompletedCallback  
    theCompletedCallback,  
    gsi_time theTimeoutMs );
```

Routine	Required Header	Distribution
adDownloadNewCreatives	<ad.h>	SDKZIP

Return Value

This function returns `AdResult_NO_ERROR` upon success. Otherwise a valid `AdResult` error condition is returned. (see remarks)

Parameters

theInterface

[in] SDK interface previously initialized using `adInitialize`

theThrottle

[in] Bandwidth throttle in bytes

theProgressCallback

[in] Function to be called periodically with progress info

theCompletedCallback

[in] Function to be called when ALL downloads have completed

theTimeoutMs

[in] Timeout for connecting to each server (not total download time)

Remarks

adDownloadNewCreatives will download missing creative files for the registered positions. The supplied progress callback will be periodically triggered with updated status information.

In most cases, the SDK will save the creative into the ad file cache. On platforms without disk access the data will be streamed into memory. Developers on these platforms should copy the data buffer into a permanent location.

See `adQueryForActiveUnits` for how the SDK determines which creatives to download.

Section Reference: [Gamespy Advertising SDK](#)

See Also: [adQueryForActiveUnits](#), [adCancelDownloads](#)

adEndTrackUsageTime

Ends time-based usage tracking for the specified unit.

```
AdResult adEndTrackUsageTime(  
    AdInterfacePtr theInterface,  
    const char * thePositionName,  
    AdUsageCategory theCat );
```

Routine	Required Header	Distribution
adEndTrackUsageTime	<ad.h>	SDKZIP

Return Value

This function returns `AdResult_NO_ERROR` upon success. Otherwise a valid `AdResult` error condition is returned. (see remarks)

Parameters

theInterface

[in] SDK interface previously initialized using `adInitialize`

thePositionName

[in] String name of the position registered using `adRegisterPosition`

theCat

[in] UC_VIEWS or UC_INTERACTIONS

Remarks

See `adBeginTrackUsageTime` for details on usage tracking.

Section Reference: [Gamespy Advertising SDK](#)

See Also: [adBeginTrackUsageTime](#), [adIncrementUsageCount](#)

adGetUnitInfoByID

Retrieves the AdUnitInfo the specified unit.

```
AdResult adGetUnitInfoByID(  
    AdInterfacePtr theInterface,  
    AdUnitID theUnitID,  
    AdUnitInfo ** theInfoOut );
```

Routine	Required Header	Distribution
adGetUnitInfoByID	<ad.h>	SDKZIP

Return Value

This function returns `AdResult_NO_ERROR` upon success. Otherwise a valid `AdResult` error condition is returned. (see remarks)

Parameters

theInterface

[in] SDK interface previously initialized using adInitialize

theUnitID

[in] A valid Unit ID

theInfoOut

[out] A pointer to the unit's info. Valid until the next call to adThink.

Remarks

In most cases `adGetUnitInfoByPosition` should be used. This will be the default unit information if either `adQueryForActiveUnits` or `adDownloadNewCreatives` has not completed.

Section Reference: [Gamespy Advertising SDK](#)

See Also: [adGetUnitInfoByPosition](#), [adQueryForActiveUnits](#), [adDownloadNewCreatives](#)

adGetUnitInfoByPosition

Retrieves the AdUnitInfo for the unit in the specified position.

```
AdResult adGetUnitInfoByPosition(  
    AdInterfacePtr theInterface,  
    const char * thePositionName,  
    AdUnitInfo ** theInfoOut );
```

Routine	Required Header	Distribution
adGetUnitInfoByPosition	<ad.h>	SDKZIP

Return Value

This function returns `AdResult_NO_ERROR` upon success. Otherwise a valid `AdResult` error condition is returned. (see remarks)

Parameters

theInterface

[in] SDK interface previously initialized using `adInitialize`

thePositionName

[in] String identifier of the position registered with `AdRegisterPosition`

theInfoOut

[out] A pointer to the unit's info. Valid until the next call to `adThink`.

Remarks

This will be the default unit information if either `adQueryForActiveUnits` or `adDownloadNewCreatives` has not completed.

Section Reference: [Gamespy Advertising SDK](#)

See Also: [adRegisterPosition](#), [adQueryForActiveUnits](#), [adDownloadNewCreatives](#), [adGetUnitInfoByID](#)

adIncrementUsageCount

Increment ad usage (without tracking time in use).

```
AdResult adIncrementUsageCount(  
    AdInterfacePtr theInterface,  
    const char * thePositionName,  
    AdUsageCategory theCat );
```

Routine	Required Header	Distribution
adIncrementUsageCount	<ad.h>	SDKZIP

Return Value

This function returns `AdResult_NO_ERROR` upon success. Otherwise a valid `AdResult` error condition is returned. (see remarks)

Parameters

theInterface

[in] SDK interface previously initialized using `adInitialize`

thePositionName

[in] String name of the position registered with `adRegisterPosition`

theCat

[in] UC_VIEWS or UC_INTERACTIONS

Remarks

This function increments the usage count for the specified ad. Use this function when tracking non time-based interactions such as drinking a soda. See `adBeginTrackUsageTime` for a complete description of usage tracking.

Section Reference: [Gamespy Advertising SDK](#)

See Also: [adBeginTrackUsageTime](#), [adEndTrackUsageTime](#)

adInitialize

Create an SDK instance.

```
AdResult adInitialize(  
    const AdInitParams* theInitParams,  
    AdInterfacePtr * theInterfaceOut );
```

Routine	Required Header	Distribution
adInitialize	<ad.h>	SDKZIP

Return Value

This function returns `AdResult_NO_ERROR` upon success. Otherwise a valid `AdResult` error condition is returned. (see remarks)

Parameters

theInitParams

[in] Structure with SDK runtime settings

theInterfaceOut

[out] AdInterfacePtr to be initialized

Remarks

This function will initialize an `AdInterfacePtr` which may then be passed to the Ad SDK interface functions.

The `AdInitParams` parameter is a structure of SDK runtime options. Please see the developer guide documentation for a description of each field.

Example

```
AdInterfacePtr anInterface = NULL;
AdResult       aResult     = AdResult_NO_ERROR;

    // Set run-time parameters
    AdInitParams anInitParams;
    memset(&anInitParams;, 0, sizeof(AdInitParams));
    anInitParams.mGameId = GAME_ID;
anInitParams.mOfflineFilePath = "ad";

    printf("Initializing the Ad SDK\r\n");
    aResult = adInitialize(&anInitParams;, &anInterface;
    if (aResult != AdResult_NO_ERROR)
    {
        printf("adInitialize failed (%d)\r\n", aResu
        return 0;
    }
```

Section Reference: [Gamespy Advertising SDK](#)

See Also: [adRegisterPosition](#), [adThink](#), [adShutdown](#)

adQueryForActiveUnits

Queries the list of active ad units from the ad server.

```
AdResult adQueryForActiveUnits(  
    AdInterfacePtr theInterface,  
    gsi_u32 theProfileId,  
    gsi_u32 theSex,  
    gsi_u32 theBirthDay,  
    gsi_u32 theBirthMonth,  
    gsi_u32 theBirthYear,  
    char* theCountryCode,  
    AdQueryForActiveUnitsCallback theCallback,  
    gsi_time theTimeoutMs );
```

Routine	Required Header	Distribution
adQueryForActiveUnits	<ad.h>	SDKZIP

Return Value

This function returns `AdResult_NO_ERROR` upon success. Otherwise a valid `AdResult` error condition is returned. (see remarks)

Parameters

theInterface

[in] SDK interface previously initialized using `adInitialize`

theProfileId

[in] Profileid of the local user. Usually obtained via the GP SDK.

theSex

[in] Usually obtained via the GP SDK.

theBirthDay

[in] Usually obtained via the GP SDK.

theBirthMonth

[in] Usually obtained via the GP SDK.

theBirthYear

[in] Usually obtained via the GP SDK.

theCountryCode

[in] Two letter country code. Usually obtained via the GP SDK.

theCallback

[in] Callback to be triggered when the operation completes.

theTimeoutMs

[in] Timeout in milliseconds

Remarks

This function will retrieve information about one advertisement for each registered position. Advertisements will not be downloaded until `adDownloadNewCreatives` is called.

Supplied user data will be used for targeted advertising.

Section Reference: [Gamespy Advertising SDK](#)

See Also: [adGetUnitInfoByPosition](#), [adDownloadNewCreatives](#)

adRegisterPosition

Registers the string name and default ad of a new position.

```
AdResult adRegisterPosition(  
    AdInterfacePtr theInterface,  
    const char* thePositionName,  
    AdUnitID theDefaultAdId,  
    const char* theDefaultAdLocalResourceName,  
    const char* theDefaultAdExtraData );
```

Routine	Required Header	Distribution
adRegisterPosition	<ad.h>	SDKZIP

Return Value

This function returns `AdResult_NO_ERROR` upon success. Otherwise a valid `AdResult` error condition is returned. (see remarks)

Parameters

theInterface

[in] SDK interface previously initialized using `adInitialize`

thePositionName

[in] String name of the position registered through the web interface.

theDefaultAdId

[in] AdID for tracking usage statistics.

theDefaultAdLocalResourceName

[in] Filename for the advertisement

theDefaultAdExtraData

[in] Developer defined data to be used client side only.

Remarks

The string name of the position must match the name created using the Ad Portal web interface. The default ad information will be returned by the SDK if no other advertisement is available. Default advertisements should use a registered AdID for reporting. This will allow the SDK to report "missed" usage opportunities.

Section Reference: [Gamespy Advertising SDK](#)

See Also: [adGetUnitInfoByPosition](#), [adGetUnitInfoByID](#)

adSendUnitUsageData

Uploads usage data to the Ad Server.

```
AdResult adSendUnitUsageData(  
    AdInterfacePtr theInterface,  
    AdSendUnitUsageDataCallback theCallback,  
    gsi_time theTimeoutMs );
```

Routine	Required Header	Distribution
adSendUnitUsageData	<ad.h>	SDKZIP

Return Value

This function returns `AdResult_NO_ERROR` upon success. Otherwise a valid `AdResult` error condition is returned (see remarks).

Parameters

theInterface

[in] SDK interface previously initialized using `adInitialize`

theCallback

[in] Function to be triggered when the operation completes

theTimeoutMs

[in] Timeout for the operation, in milliseconds

Remarks

Win32 Specific: If the upload fails for any reason, the usage data will be stored in an offline file and sent the next time **adSendUnitUsageData** is called.

Section Reference: [Gamespy Advertising SDK](#)

See Also: [adBeginTrackUsageTime](#), [adEndTrackUsageTime](#), [adIncrementUsageCount](#)

adShutdown

Destroys the SDK interface object and frees any allocated memory.

```
AdResult adShutdown(  
    AdInterfacePtr theInterface );
```

Routine	Required Header	Distribution
adShutdown	<ad.h>	SDKZIP

Return Value

This function returns `AdResult_NO_ERROR` upon success. Otherwise a valid `AdResult` error condition is returned. (see remarks)

Parameters

theInterface

[in] SDK interface previously initialized using `adInitialize`

Remarks

Calling this function will release the AdInterfacePtr and free all internally allocated memory. The AdInterfacePtr can no longer be used by the SDK.

AdResult_INVALID_PARAMETERS will be returned in theInterface is invalid.

Section Reference: [Gamespy Advertising SDK](#)

See Also: [AdInitialize](#)

adThink

Allows the SDK to continue processing.

```
AdResult adThink(  
    AdInterfacePtr theInterface );
```

Routine	Required Header	Distribution
adThink	<ad.h>	SDKZIP

Return Value

This function returns `AdResult_NO_ERROR` upon success. Otherwise a valid `AdResult` error condition is returned. (see remarks)

Parameters

theInterface

[in] SDK interface previously initialized using `adInitialize`

Remarks

This function should be called as frequently as possible to allow the SDK to continue processing. SDK callbacks will be dispatched during this function call.

AdResult_INVALID_PARAMETERS will be returned if *theInterface* is invalid.

Section Reference: [Gamespy Advertising SDK](#)

ATLAS Competition SDK

Overview

Most developers would like to add competition to the multiplayer portion of their game, but they don't always have the time or resources to create such a complex system. The GameSpy Competition SDK is an easy-to-use solution that enables developers to insert competition functionality into their games, while maintaining a high level of flexibility and security.

You might be thinking, what is competition? By definition, competition involves striving to achieve dominance or attaining a set goal. In games, this generally refers to the inclusion of developer-defined metrics used to measure a player's performance and compare these stats against others. This could be in the form of a Leader Board which tracks the top players with a given category, such as who has the top all-time win/loss ratio. It can also be in the form of individual accomplishments that tell a player how well they have done in a given area, such as weapon accuracy or percentage of the game completed (for single player games). The Competition SDK allows developers to pick and choose what they deem necessary as statistics by which players can compare themselves to their peers.

The Competition SDK provides an easy web interface in order to customize the statistics and data reported to our backend. Since everything is setup using this simple Administrative site, there is no scripting or custom code required in order to process this data; this is all accomplished via the web interface. The Competition API allows you to send game results to our central servers, which will then go through all received reports and *normalize* these to create a final report. The normalization process handles any discrepancies that might arise (and applying penalties where necessary) in order to establish an official final report it considers to be the most accurate representation of what took place. Statistics are then stored in our generalized remote storage system called Sake; the game must utilize the GameSpy Sake SDK in order to retrieve player stats.

Features

- Easy-to-use web interface (Admin Site) to create keys/rules for a game.
- Custom Game-specific data or **keys** submitted in addition to the game results.
- Custom **rules** that are used to process raw statistical data
- Final **stats** are stored in Sake storage system

Admin Site (ATLAS web interface)

The Administrative site (<http://tools.gamespy.net/atlas/>) will allow for creation of keys, processing rules, and stats. If you do not have access to the site, you can gain access by e-mailing devsupport@gamespy.com. The site contains numerous examples designed to help guide you along in creating the necessary elements for adding stats in your game. If you have problems with the site, or any questions that are not directly addressed, please e-mail us at devsupport@gamespy.com

On the Home page of the Admin Site you will see links to create Ruleset(s) to integrate into your C/C++ codebase. These files contain defines for the KEYS, STATS, and ATLAS_RULE_SET_VERSION which can be used when building reports or retrieving stats. To see an example of this the ScRaceSample includes the auto-generated header *atlas_Compensation_Race_Sample_App_v1.h*, which is referenced in the sample code as well.

Custom Keys

Developers can create user-defined **keys** to submit custom data along with the generic game results (such as win, loss, etc.). Each key that the game reports is paired with a value that represents some relevant information from the game that is either global, player, or team specific.

Custom Processing Rules and Generating Stats

Developers will utilize the ATLAS Admin site in order to create these keys/rules/stats for their game. Custom processing **rules** are used to process **stats** via a set of input/output specifications and a selected

operation (addition, subtraction, etc.). The inputs can be either previous stats, or keys that were sent along with game results. The output is generally stored as the stats. There will also be common rules that developers can use for their stats calculation.

An example: You might want to keep track of the number of times a particular map is played. To do this you would have a single *rule*, *key*, and *stat* for each map. Then the input key for the game would be something like "boatMap/1" indicating that I just played the "boatMap" map. The rule would be an "incremental" rule *operation* taking the "boatMap" key and "numBoatMapPlayed" stat as inputs and returning "numBoatMapPlayed" as an output stat. This rule would then increment the "numBoatMapPlayed" stat as long as the "boatMap" key exists with a value of 1.

Security

The biggest threat to competition in online gaming lies in cheating; individuals who want to hack the system in order to gain an unfair advantage. As it currently stands, many players are reluctant to play in online competitions because of this. In order to prevent foul play and help retain the ideal of fair-play in online gaming, the Competition SDK provides a built-in solution to ensure the accuracy and the validity of its results.

Authentication/Validation of Players

The first piece of the puzzle is the use of the GameSpy Authentication service, which validates players. It allows the Competition backend to verify a player is really who he claims to be. Each player involved in a game must obtain a certificate that will be sent along with the results to the Competition service. The certificate will act as a signature of that player, ensuring that the results being submitted are not from an unknown source.

Authoritative v. Collaborative Reports

Next, the SDK allows games to send **authoritative** reports or

collaborative reports. An **authoritative** report is the equivalent of an official referee-type report of what occurred during gameplay, therefore there must be at least one authoritative report per session that contains the results for all players and teams. A **collaborative report** contains unofficial data in order to collaborate with the other data being submitted. Unlike an authoritative report, these reports are not as restricted and may contain more information that may be specific to each player. For more information about how to best organize these reports sent, see Appendices II-III below for use-case scenarios regarding typical game types and the reports submitted.

Report Encryption

Currently, the SDK has SSL encryption for encrypting its report submissions. While we feel this is sufficient as a starting basis (in addition to the other securities used in the SDK), report encryption is still a work in progress and other options are being explored to provide further enhanced security.

Tips for Host Migration

For Host migration, the only difference will be that the newly chosen host will change their **intention** mid-game (if it is not already authoritative, see use-case scenarios below) to submit an authoritative report, as they are now considered the new referee (or the official view) of the game. Also, the Competition system works from "more is better" perspective when it comes to reports, so all players in the game should make sure to send a report in order to provide the most accurate statistics results. Note that for peer-peer games that allow Host migration & Late Entry, this is very specific scenario covered below in Appendix V.

Normalization of Reports

The Competition normalizes reports in order to analyze the game results and provide the best possible explanation for what took place. The normalizer will:

- handle irregularities, such as disconnects or host migration

- handle discrepancies between reported results

Getting Started

Here is a quick rundown of how the process works. Data is submitted as a report to the backend in terms of key/value pairs where the *Keys* have been predefined on the Admin Site and referenced via the Key ID. The backend processes this data based on the *Rules* you have defined and places this output into a Stat. These *Stats* are then generated as records within the Sake database, corresponding to the appropriate table for the *Rule Type* (ie. GameStats_vX, PlayerStats_vX, TeamStats_vX, or StaticStats_vX) - see Appendix I for more information about retrieving stats.

The *Rule Type* is very specific to the type of Stats being stored:

GameStats - Game-specific data, not related to any specific player or to any specific team. Because of this, you will equivalently have 1 record in this table, corresponding to the overall game. The type of Stats stored here could be something like the number of times a given map has been played, or the aggregate total of bullets fired in the game (from all players).

PlayerStats – Player-specific data. 1 record per player. This contains any stats related directly to that player, such as number of overall kills, average kills, average wins, losses, total wins, losses, number of bullets fired, etc.

TeamStats - Team-specific data. 1 record per team.

StaticStats - global static data used in processing other Stats. 1 record in this table for all Static stats.

Let's say you want to create a PLAYER_HIGH_SCORE. Here's how:

- First you would need to create the Key which would indicate the data submitted in a report to the system. In this case, the key would be the player's score. Let's call this key KEY_SCORE.
- In your implementation of the SDK, the report would submit data for each player's score during the match. The key ID here must match

the ID for the KEY_SCORE key in order to indicate the value submitted corresponds to this key. You can generate a Ruleset Header File from the main page of the Admin site (at the bottom) which contains defines for these Key IDs you can reference in your game. For example, if KEY_SCORE had ID #1, then the define would be `#define KEY_SCORE 1` so that your code would reference the KEY_SCORE when adding data to the report.

- Next, a Stat needs to be created which is used to hold the Statistic generated from the processing rule. Let's call this STAT_HIGH_SCORE.
- This STAT_HIGH_SCORE is generated via a rule that we need to create. So we can make a RULE_HIGH_SCORE, which is a Player-type rule that takes KEY_SCORE and STAT_HIGH_SCORE as input, applies the Maximum operation to the inputs, and outputs to STAT_HIGH_SCORE. The Player type rule indicates that this is a per-player based Statistic.
- Once you a report has been submitted and processed, a record will be generated in the PlayerStats_vX (where X indicates the ruleset version number) table in Sake containing your Stat. See Appendix I below to see how you can retrieve this data via the Sake SDK.

Dependencies

The Competition SDK is dependent upon the following GameSpy SDKs - in order to use the Competition SDK you must also include these packages. The latest versions of these files are available from <http://www.gamespy.net/secure/download/>.

Common Code

The Competition SDK uses the GameSpy Common Code package. Once you have both this package and the Competition package, both need to be extracted into a single directory where all GameSpy SDKs can be stored and easily referenced. An example directory structure might look like the following:

```
\Gamespy
  \common
  \sc
  \webservices
```

GameSpy HTTP SDK

The Competition SDK uses this SDK to send requests and receive responses. It is important to have this SDK along with the common code.

GameSpy Authentication Service

Authentication services are used to obtain login certificates for the Competition SDK. These files are included with the Competition SDK download and located in the *webservices* folder.

GameSpy Sake SDK

Games must use the Sake SDK to retrieve the stats the Competition system stores in the Sake backend.

SDK Implementation

Before using Competition, a game must have first performed the standard GameSpy Availability Check. This ensures that the GameSpy backend is available, and that the current game has access to the backend. If the game has not performed the availability check prior to initializing the SDK, the call to `scInitialize` will return `SCResult_NO_AVAILABILITY_CHECK`.

This section explains the necessary steps in order to implement the basics of the Competition SDK. The following is a brief summary of the steps for implementation:

- The SDK and all dependent components are initialized, so they are ready for use
- Player's login. Game authenticates each player via the auth service and retrieves a login certificate used by the SDK to prove a player's authenticity.
- Host creates a game session and distributes
- All players set their report intentions, which describes the type of report (*authoritative* or *collaborative*) being submitted.
- Gameplay begins, stats are recorded.
- When the game session is complete, all players create a report with the stats recorded. This report contains (i) global, (ii) player, and (iii) team data, submitted in that order.
- The reports are submitted to the Competition Backend, to be processed according to the rules setup using the Admin site.

1. Initialization

Before doing anything with the Competition SDK itself, you must start the GameSpy core using `gsCoreInitialize`. The core allows the Competition SDK to initiate and complete its tasks. The authentication service included with the SDK will also require the core to be initialized before use.

```
void gsCoreInitialize();
```

Once the core has started, initialize the Competition SDK using `scInitialize`. The function will return a `SCResult` for error checking. The actual object that the game needs to keep track of is the `SCInterfacePtr`. Most functions (except those with Report in the name) will require the `SCInterfacePtr` for their corresponding operations or retrieval of data.

```
SCResult scInitialize
(
    int          theGameId,          //GA
    SCInterfacePtr * theInterfaceOut //pc
);
```

2. Login and Authentication

Before creating a session or submitting reports, players will need to login and authenticate themselves via the auth service included in the SDKs. To login under a GameSpy Presence (GameSpy ID) account the game should call `wsLoginProfile`. If the game has not performed the standard GameSpy Availability Check prior to this login attempt, it will fail with a result of `WSLogin_NoAvailabilityCheck`.

```
gsi_u32 wsLoginProfile
(
    int          partnerCode,
    int          namespaceId,
    const char * profileNick,    //profile ni
    const char * email,         //email addr
    const char * password,      //password a
    const char * cdkeyhash,     //cdkey hash
    WSLoginCallback callback,   //the callba
    void *       userData       //optional u
);
```

`partnerCode`

The **partnerid** assigned to you by GameSpy (note: not all games

use a separate partnerspace). For most games, this will use the generic GameSpy partnerspace, [WSLogin_PARTNERCODE_GAMESPY](#).

namespaceId

The **namespaceid** assigned to you by GameSpy (note: not all games use a separate namespace). If your game does not use unique nicks, you can use [WSLogin_NAMESPACE_SHARED_NONUNIQUE](#). For games using the GameSpy shared default unique nick namespace, use [WSLogin_NAMESPACE_SHARED_UNIQUE](#).

Within the login callback, the [WSLoginResponse](#) object will contain the login certificate and private data which will be passed to subsequent Competition SDK calls. These values should be stored for later use:

- [GSLoginCertificate](#) mCertificate
- [GSLoginPrivateData](#) mPrivateData

3. Creating a session

Once the Host has completed his login, he can create a session using the login certificate and private data mentioned above. A session is generally created for each unique game instance (i.e. a match with a clear winner/loser or end criteria), but is not limited to this:

```
SCResult scCreateSession
(
    SCInterfacePtr                theInterface,
    const GSLoginCertificate *    theCertificate,
    const GSLoginPrivateData *   thePrivateData,
    SCCreateSessionCallback      theCallback,
    gsi_time                     theTimeoutMs,
    void *                       theUserData
);
```

This function will send request to the Competition backend to create a session. If the result of this call is anything other than

`SCResult_NO_ERROR`, this indicates an error has occurred. Once a session has been created, the host can retrieve the session ID by calling `scGetSessionId`:

```
const char * scGetSessionId(const SCInterfacePtr the
```

The host will then need to distribute this session ID to the clients. Each client will then set his session ID with the SDK using `scSetSessionId` before setting his report intention. The session ID has a constant length of `SC_SESSION_GUID_SIZE`.

```
SCResult SC_CALL scSetSessionId
(
    const SCInterfacePtr theInterface,
    const gsi_u8          theSessionId[SC_SESS
);
```

Note that once a session is created, the backend begins a countdown. If a report for a designated session has not been received within 10 hours from creation, the session times out and is no longer valid. Once the first report has been received, this timeout period decreases to 2 minutes and reports will timeout if not received before this limit expires. Note that a session will never need to be explicitly cancelled as it will eventually timeout if no reports for it have been received.

4. Setting the Report Intention

At this point, the host and clients need to set their report intentions. These intentions should be set by all players submitting a report, prior to starting gameplay.

As a general rule of thumb, the current host will always submit an **authoritative** report and the clients will submit **collaborative** reports - this is done by flagging the `isAuthoritative` parameter when setting intention. However, depending on the game type (i.e. RTS versus FPS) being played, players may set different intentions for themselves. In addition, if Host Migration is involved, these intentions may change mid-game. For more specifics on this, please see Appendices II-III. Also,

players can set multiple intentions in order to send multiple reports - a scenario for why this might be used is described in the use-case examples below:

```
SCResult scSetReportIntention
(
    const SCInterfacePtr    theInterface,
    const gsi_u8            theConnectionID,
    gsi_bool                isAuthoritative,
    const GSLoginCertificate * theCertificate,
    const GSLoginPrivateData * thePrivateData,
    SCSetReportIntentionCallback theCallback,
    gsi_time                theTimeoutMs,
    void *                  theUserData
);
```

After setting his intention, each player should retrieve his connection ID (unless returning to a match and using their previous one) which will be used upon submitting a report. This is done by calling [scGetConnectionId](#). The one caveat here is that the host, or any player submitting an authoritative snapshot, will be reporting data for all players and thus needs to store the connection IDs for all players in addition to their own. Players should therefore exchange connection IDs with one another before beginning play.

```
const char * scGetConnectionId(const SCInterfacePtr
```

Once all intentions have been set, gameplay can begin and the game should begin recording stats for that game session.

5. Creating the Report

Once the game session is complete, everyone who participated in the game should submit a report. To do this, each player will first need to create a report object by calling [scCreateReport](#). This report should be created at the end of a game session to ensure the most accurate values for the player/team count. Note that [theHeaderVersion](#) parameter corresponds to the [ATLAS_RULE_SET_VERSION](#) located in the auto-

generated from the Admin site. This header file can be retrieved after keys for the game have been created on the Admin site (click "Download Ruleset Header File" on the home page for your game).

```
SCResult scCreateReport
(
    const SCInterfacePtr    theInterface,
    gsi_u32                 theHeaderVersion,
    gsi_u32                 thePlayerCount,
    gsi_u32                 theTeamCount,
    const SCReportPtr *    theReportOut
);
```

The game should keep track of `theReportOut` report object for adding stats in key/value pairs to the report. This is done in three stages, for each type of data: global, player, and team data. This should be done in the order shown below so that errors do not occur; **(a) Global**, **(b) Player**, and finally **(c) Team data**:

a. Global Data

Before submitting each type of data, the game must first inform the competition SDK of the data it is about to report. This is done by calling the appropriate `scReportBegin*` function before submitting this type of data to the report. For global data this function is:

```
SCResult scReportBeginGlobalData(SCReportPtr theRepc
```

After this call is made, the game will submit its data by calling either `scReportAddIntValue` or `scReportAddStringValue`.

```
SCResult scReportAddIntValue
(
    SCReportPtr theReportData,    //pointer to
    gsi_u16 theKeyId,            //the key va
    gsi_i32 theValue              //the value
);
```

```
SCResult scReportAddStringValue
(
    SCReportPtr theReportData,
    gsi_u16 theKeyId,
    const gsi_char * theValue
);
```

b. Player Data

Next the game will submit player data. First the game should notify the Competition SDK it plans to report player data by calling `scReportBeginPlayerData`:

```
SCResult scReportBeginPlayerData(SCReportPtr theRepc
```

The game will indicate each new player to be reported by calling `scReportBeginNewPlayer`:

```
SCResult scReportBeginNewPlayer(SCReportPtr theRepor
```

The game will then call `scReportSetPlayerData` to set the initial data for this new player to be reported. The connection ID (retrieved via `scGetConnectionId`) is passed to the `thePlayerConnectionId` parameter in order to designate the player whose stats are being reported. The `theResult` parameter is an enumerated value that describes the final game result for the given player.

Note: `theAuthData` is currently unused in this version of the SDK.

```
SCResult scReportSetPlayerData
(
    SCReportPtr theReport,
    gsi_u32 thePlayerInd
    const gsi_u8 thePlayerCon
    gsi_u32 thePlayerTea
```

```

        SCGameResult          theResult,
        gsi_u32               theProfileId,
        const GSLoginCertificate * theCertificate,
        const gsi_u8          theAuthData[
    );

```

After this call is made, the game should report player-specific key/value data by calling either [scReportAddIntValue](#) or [scReportAddStringValue](#) as it applies. Once the game has finished reporting data for a specific player, if it has more player data to report, it should begin this process again starting with [scReportBeginNewPlayer](#) and repeating the above steps, until data for all players has been submitted.

c. Team Data

Lastly, the game should notify the SDK that is about to report team data:

```

SCResult scReportBeginTeamData(SCReportPtr theReport

```

Just as with player data, the game will then call the team equivalent [scReportBeginNewTeam](#) followed by [scReportSetTeamData](#) to tell the SDK which team data is about to be reported. The [theResult](#) parameter is an enumerated value that describes the final game result for the given team.

```

SCResult scReportBeginNewTeam(SCReportPtr theReport

```

```

SCResult scReportSetTeamData
(
    SCReportPtr    theReport,
    gsi_u32        theTeamIndex,
    SCGameResult   theResult
);

```

Once the report is complete and ready for submittal, the game should call `scReportEnd` to indicate its completion.

```
SCResult scReportEnd
(
    SCReportPtr    theReport,
    gsi_bool       isAuth,           //gsi_true f
    SCGameStatus   theStatus        //enum descr
);
```

For `SCGameStatus` reporting, the game should do the following. As long as the game finished properly, and no one disconnected during the course of play, then all players in the match should submit `SCGameStatus_COMPLETE` reports. If any members disconnected during play, but the game was finished completely, then all players in the match should submit `SCGameStatus_PARTIAL` reports indicating that disconnects occurred. For any players who do not complete the match, a `SCGameStatus_BROKEN` report should be submitted. Thus if the game did not completely finish, all players will submit broken reports. The only case that will trigger an invalid report is if reports for the same game describe status as both `SCGameStatus_COMPLETE` and `SCGameStatus_PARTIAL`. Since `COMPLETE` indicates that all players finished the game w/o a disconnect and `PARTIAL` indicates that disconnects occurred, at no time should a game report both complete and partial - this will be seen as an exploit and invalidate the report.

For a better example of the report submission process, see the Competition SDK sample application and Appendices II-III for game type specific usage scenarios.

6. Submitting the Report

Once a report has been completed with a call to `scReportEnd`, the game should then submit the report to the Competition backend by calling:

```
SCResult scSubmitReport
(
```

```

const SCInterfacePtr    theInterface
const SCReportPtr      theReport,
gsi_bool                isAuthoritat
const GSLoginCertificate * theCertifica
const GSLoginPrivateData * thePrivateDa
SCSubmitReportCallback theCallback,
gsi_time                theTimeoutMs
void *                  theUserData
);

```

For sending an authoritative support the game should pass `gsi_true` to the `isAuthoritative` parameter. This should match the intention that the player set (refer to `scSetReportIntention`). If the result of this call is anything other than `SCResult_NO_ERROR`, this indicates an error has occurred.

Once the report has been submitted, the backend will send back the result of the submission via the `SCSubmitReportCallback`. This callback will indicate to the game if any errors during the submission. An invalid certificate or invalid private data will cause the operation to fail. An incomplete or empty report will also cause this operation to fail.

7. Thinking

All interface functions that have callbacks will require the game to call `scThink`. In addition the value returned by this function should also be checked in case any problems occur:

```

SCResult scThink(SCInterfacePtr theInterface);

```

Remember to call this function in the main loop. All SDK calls should be made from within the same thread. See the following Knowledge Base entry for more information: [Are your SDKs thread-safe?](#)

The authentication service requires a call to the Gamespy Core think function since it uses a different service for authentication:

```

void gsCoreThink(gsi_time theMs);

```

8. Shutting Down

Shutting down the Competition SDK is done using the following, it will take care of cleaning up resources used by the SDK:

```
SCResult scShutdown(SCInterfacePtr theInterface);
```

In addition, the game should clean up the Gamespy core by using the function:

```
void gsCoreShutdown();
```

Appendix I: Retrieving Stats

The Competition SDK only reports gamedata to the backend, it does not retrieve it. When stats are created via the reported results they are stored into the Sake database. The GameSpy Sake SDK is used to retrieve Stats. Please refer to the Sake documentation for more in-depth information about implementing the Sake SDK.

To begin, developers can access the Sake Admin website at <http://tools.gamespy.net/SakeAdmin/>. After selecting your game, you will be brought to a page to see your game's tables where the stats have been stored. Tables in Sake are automatically generated via the ATLAS Administration site when you create keys for your game. These tables are **GameStats_vX**, **PlayerStats_vX**, **TeamStats_vX**, and **StaticStats_vX** which you can see under the Sake Admin page for your game. The vX refers to the version number used in both the Admin site as well as when integrating ATLAS (for example, version 1 for a game will have tables defined as GameStats_v1, etc.).

Clicking on "Fields" for any of these tables will show a list of fields which indicate a given Stat. Note that the Stats that are generated from the Rules setup on the ATLAS Admin site will automatically be recorded into these generated Tables, based on the Rule type set when creating the rule. In other words, Game STATS which are generated are stored in the GameStats table, as Player STATS are stored in the PlayerStats table, and so on. You can then retrieve this data via Sake by querying these tables with the field names that correspond to the STATS created on the ATLAS Admin site. The first time you submit a session and the rules process this data into stats, you should see these fields created.

You will use these **TableIds** and **FieldNames** in Sake calls to retrieve stats in your game. The easiest method to do so is to use the Sake call [sakeSearchForRecords](#). You can search in a given table, across various fields using an SQL-like filter string along with sorting criteria.

For example, let's say you want to order and show the "top 100-200 entries in descending order of player high scores that are > 50000". You have created a stat called "PLAYER_HIGH_SCORE" which contains a

player's current high score. In addition, you have defined a player-type rule in order to calculate this PLAYER_HIGH_SCORE stat; this is all done with ATLAS ruleset version 1. Therefore, after this stat is calculated it will generate a record in the "PlayerStats_v1" table for your game. You would then search for records where the [SAKESearchForRecordsInput](#) has the following values:

```
mTableID = "PlayerStats_v1"  
mFieldNames = "PLAYER_HIGH_SCORE", "ownerid"  
mNumFields = 2  
mFilter = "PLAYER_HIGH_SCORE > 50000"  
mSort = "PLAYER_HIGH_SCORE desc"  
mOffset = "100"  
mMaxRecords = "100"
```

This would retrieve the result you seek. Each record returned would be that of a player's high score, the player of which is identified based upon the ownerid (owner's profileid) of the given record.

Please also refer to Appendix IV in the SAKE Overview for more details about specific Leaderboard queries and optimizations (e.g. getting a player's rank, etc.).

Appendix II: Use Case - Real-Time Strategy (RTS) Game

The following describes the recommended approach for report submissions with RTS-style games. This is specifically referring to peer-peer games that do not have late entry and may or may not allow Host Migration. The primary difference between this game type and those of dedicated server games is that no single player is really the authoritative view of the game. Since the game is by definition peer-peer, all players essentially act as an official voice of what transpired.

This being the case, we recommend that **ALL players submit authoritative reports**. This takes care of two common problems. First off, it eliminates the possibility of a 1v1 match where no authoritative report is sent if the host disconnects. By having all players submit authoritative reports, we can ensure that if a host disconnects in a heads-up match, the opponent will report this disconnect as well as this player's data. Secondly, it takes care of any Host Migration issues automatically. By having each player submit an authoritative report, there is no need for players to change their intentions mid-game.

Appendix III: Use Case - First Person Shooter (FPS) Game

Unlike an RTS-style game, the FPS game type we are describing here is that of a dedicated server game allowing late entry. The difference here is that the server itself acts as a dedicated host or official view of gameplay. Since these games allow late entry, it's common to have players joining/leaving/disconnecting during the course of gameplay. Thus, we need to have a single official view of the overall game that can monitor all of the activity. This does not necessarily mean that the host will submit *ALL* data for every player in the game, as this could grow immensely large over a long game. It simply means the host will need to be the final say for players that disconnect, or the host will corroborate a player's collaborative report with his own.

Therefore **the host will submit an authoritative report and all clients will ONLY submit collaborative reports**. This means there will only be a single authoritative report per-session (map change, round, etc.), submitted from the host. This authoritative report should only initially contain the host's player data (for non-dedicated hosts) and the game results. In addition to this, **the host should ALSO submit collaborative reports for all late-entry players**. Doing so will corroborate a player's reported statistics. The only caveat here is that if this player unexpectedly disconnects before the game is complete, we want to ensure their data is reported. To account for this, **should a player disconnect unexpectedly, the host will submit that player's data as part of his authoritative report INSTEAD of sending a collaborative report for that player**. Doing so will ensure this player's data is recorded even when this player is disconnected from the game. Player's that disconnect normally (in other words, they forced a disconnect by quitting out of the game) should report their collaborative snapshots themselves. This is a recommended approach for any intentional disconnect as illustrated in the Competition Sample Application.

Appendix IV: Troubleshooting / FAQ

The GameSpy Knowledge Base is kept up-to-date with important troubleshooting tips and information about the Competition SDK:
<http://www.poweredbygamespy.com/secure/kb/categories.php?categoryid=10>.

Competition SDK Functions

scCreateMatchlessSession	This is a variation of scCreateSession that creates a "matchless" session; "matchless" means incoming data will be scrutinized less, and applied to stats immediately instead of when the match is over.
scCreateReport	Creates a new report for the game session
scCreateSession	Requests the Competition service to create a session ID and keep track of the session that is about to start.
scDestroyReport	Used to clean up and free the report object after it has been submitted.
scGetConnectionId	Used to obtain a Connection ID when setting player data in the report.
scGetSessionId	Used to obtain the session ID for the current game session.
scInitialize	Initializes the competition SDK.
scReportAddByteValue	Adds a byte value to the report for a specific key.
scReportAddFloatValue	Adds a float value to the report for a specific key.

scReportAddIntValue	Adds an integer value to the report for a specific key.
scReportAddShortValue	Adds a short value to the report for a specific key.
scReportAddStringValue	Adds a string value to the report for a specific key.
scReportBeginGlobalData	Tells the competition SDK to start writing global data to the report.
scReportBeginNewPlayer	Add a new player to the report
scReportBeginNewTeam	Adds a new team to the report.
scReportBeginPlayerData	Tells the competition SDK to start writing player data to the report.
scReportBeginTeamData	Tells the competition SDK to start writing player data to the report.
scReportEnd	Denotes the end of a report for the report specified.
scReportSetAsMatchless	Called after creating the report to set it as a matchless report - this is needed if the report is being submitted to a "matchless" game session.

scReportSetPlayerData	Sets initial player data in the report specified
scReportSetTeamData	Sets the initial team data in the report specified.
scSetReportIntention	Called to tell the backend the type of report that the player or host will send.
scSetSessionId	Used to set the session ID for the current game session.
scShutdown	Shuts down the Competition SDK
scSubmitReport	Initiates the submission of a report
scThink	Called to complete pending operations for functions with callbacks.

scCreateMatchlessSession

This is a variation of `scCreateSession` that creates a "matchless" session; "matchless" means incoming data will be scrutinized less, and applied to stats immediately instead of when the match is over.

```
SCResult scCreateMatchlessSession(  
    SCInterfacePtr theInterface,  
    const GSLoginCertificate * theCertificate,  
    const GSLoginPrivateData * thePrivateData,  
    SCCreateSessionCallback theCallback,  
    gsi_time theTimeoutMs,  
    void * theUserData );
```

Routine	Required Header	Distribution
<code>scCreateMatchlessSession</code>	<code><sc.h></code>	SDKZIP

Return Value

Enum value used to indicate the specific result of the request. This will return `SCResult_NO_ERROR` if the request completed successfully.

Parameters

theInterface

[in] A valid SC Interface Object

theCertificate

[in] Certificate obtained from the auth service.

thePrivateData

[in] Private Data obtained from the auth service.

theCallback

[in] The callback called when create session completes.

theTimeoutMs

[in] Timeout in case the create session operation takes too long

theUserData

[in] User data for use in callbacks. Note that it is a constant pointer in the callback

Remarks

Reports sent for matchless sessions should be marked as such using "scReportSetAsMatchless".

Section Reference: [Gamespy Competition SDK](#)

See Also: [scReportSetAsMatchless](#), [SCCreateSessionCallback](#), [scInitialize](#)

scCreateReport

Creates a new report for the game session.

```
SCResult scCreateReport(  
    const SCInterfacePtr theInterface,  
    gsi_u32 theHeaderVersion,  
    gsi_u32 thePlayerCount,  
    gsi_u32 theTeamCount,  
    const SCReportPtr * theReportOut );
```

Routine	Required Header	Distribution
scCreateReport	<sc.h>	SDKZIP

Return Value

Enum value used to indicate the specific result of the request. This will return `SCResult_NO_ERROR` if the request completed successfully.

Parameters

theInterface

[in] A valid SC Interface Object

theHeaderVersion

[in] Header version of the report

thePlayerCount

[in] Player count for allocating enough resources and verification purposes

theTeamCount

[in] Team count for allocating enough resources and verification purposes

theReportOut

[ref] The pointer to created SC Report Object

Remarks

There should have been a call to `CreateSession` and `SetReportIntention` before calling this function. This function should be called after a game session has ended. The player count and team count are more accurate at that point for dedicated server games. This function should also be called before calling any `scReport*` function. The header version can be obtained from the administration site where the keys are created. See the overview on obtaining access or send a request devsupport@gamespy.com.

Section Reference: [Gamespy Competition SDK](#)

See Also: [scReportBeginGlobalData](#), [scReportBeginPlayerData](#), [scReportBeginTeamData](#), [scReportBeginNewPlayer](#), [scReportSetPlayerData](#), [scReportBeginNewTeam](#), [scReportSetTeamData](#), [scReportAddIntValue](#), [scReportAddStringValue](#)

scCreateSession

Requests the Competition service to create a session ID and keep track of the session that is about to start.

```
SCResult scCreateSession(  
    SCInterfacePtr theInterface,  
    const GSLoginCertificate * theCertificate,  
    const GSLoginPrivateData * thePrivateData,  
    SCCreateSessionCallback theCallback,  
    gsi_time theTimeoutMs,  
    void * theUserData );
```

Routine	Required Header	Distribution
scCreateSession	<sc.h>	SDKZIP

Return Value

Enum value used to indicate the specific result of the request. This will return `SCResult_NO_ERROR` if the request completed successfully.

Parameters

theInterface

[in] A valid SC Interface Object

theCertificate

[in] Certificate obtained from the auth service.

thePrivateData

[in] Private Data obtained from the auth service.

theCallback

[in] The callback called when create session completes.

theTimeoutMs

[in] Timeout in case the create session operation takes too long

theUserData

[in] User data for use in callbacks. Note that it is a constant pointer in the callback

Remarks

The certificate and private data may be NULL if the local client is an unauthenticated dedicated server. The function should be called by the host after initializing the SDK, and obtaining a certificate and private data from the authentication service. The competition service creates and sends a session ID to the host. The callback passed in will get called even if the request failed.

Section Reference: [Gamespy Competition SDK](#)

See Also: [SCCreateSessionCallback](#), [scInitialize](#)

scDestroyReport

Used to clean up and free the report object after it has been submitted.

```
SCResult scDestroyReport(  
    SCReportPtr theReport );
```

Routine	Required Header	Distribution
scDestroyReport	<sc.h>	SDKZIP

Return Value

Enum value used to indicate the specific result of the request. This will return `SCResult_NO_ERROR` if the request completed successfully.

Parameters

theReport

[in] The pointer to a created SC Report Object.

Remarks

This should be called regardless of whether or not the report was submitted successfully. It should only be used if the report object contains a valid pointer from a successful call to `scCreateReport`.

Section Reference: [Gamespy Competition SDK](#)

See Also: [scCreateReport](#)

scGetConnectionId

Used to obtain a Connection ID when setting player data in the report.

```
const char * scGetConnectionId(  
    const SCInterfacePtr theInterface );
```

Routine	Required Header	Distribution
scGetConnectionId	<sc.h>	SDKZIP

Return Value

Parameters

theInterface

[in] A valid SC Inteface Object

Remarks

The connection id identifies a single player in a game session. It may be possible to have different connection ids during the same session since players can come and leave sessions.

Section Reference: [Gamespy Competition SDK](#)

scGetSessionId

Used to obtain the session ID for the current game session.

```
const char * scGetSessionId(  
    const SCInterfacePtr theInterface );
```

Routine	Required Header	Distribution
scGetSessionId	<sc.h>	SDKZIP

Return Value

Parameters

theInterface

[in] A valid SC Inteface Object

Remarks

The session ID identifies a single game session happening between players. After the host creates a session, this function can be called to obtain the session ID. The host can then send the session ID to all other players participating in the game session.

Section Reference: [Gamespy Competition SDK](#)

See Also: [scSetSessionId](#), [scCreateSession](#)

scInitialize

Initializes the competition SDK.

```
SCResult scInitialize(  
    int theGameId,  
    SCInterfacePtr * theInterfaceOut );
```

Routine	Required Header	Distribution
scInitialize	<sc.h>	SDKZIP

Return Value

Enum value used to indicate the specific result of the request. This will return `SCResult_NO_ERROR` if the request completed successfully.

Parameters

theGameId

[in] The Game ID issued to identify a game.

theInterfaceOut

[out] The pointer to the SC Interface Object instance

Remarks

The function must be called in order to get a valid SC Interface object. Most other interface functions depend on this interface function when being called. Note that if the standard GameSpy Availability Check was not performed prior to this call, the SDK will return `SCResult_NO_AVAILABILITY_CHECK`.

Section Reference: [Gamespy Competition SDK](#)

scReportAddByteValue

Adds a byte value to the report for a specific key.

```
SCResult scReportAddByteValue(  
    SCReportPtr theReportData,  
    gsi_u16 theKeyId,  
    gsi_i8 theValue );
```

Routine	Required Header	Distribution
scReportAddByteValue	<sc.h>	SDKZIP

Return Value

Enum value used to indicate the specific result of the request. This will return `SCResult_NO_ERROR` if the request completed successfully.

Parameters

theReportData

[in] A valid SC Report object

theKeyId

[in] Key Identifier for reporting data

theValue

[in] 8 bit Byte value representation of the data

Remarks

The host or player can call this function to add either global, player-, or team-specific data. A report needs to be created before calling this function. For global keys, this function can only be called after starting global data. For player or teams, a new player or team needs to be added.

Section Reference: [Gamespy Competition SDK](#)

See Also: [scCreateReport](#), [scReportBeginGlobalData](#), [scReportBeginPlayerData](#), [scReportBeginTeamData](#), [scReportBeginNewPlayer](#), [scReportSetPlayerData](#), [scReportBeginNewTeam](#), [scReportSetTeamData](#)

scReportAddFloatValue

Adds a float value to the report for a specific key.

```
SCResult scReportAddFloatValue(  
    SCReportPtr theReportData,  
    gsi_u16 theKeyId,  
    float theValue );
```

Routine	Required Header	Distribution
scReportAddFloatValue	<sc.h>	SDKZIP

Return Value

Enum value used to indicate the specific result of the request. This will return `SCResult_NO_ERROR` if the request completed successfully.

Parameters

theReportData

[in] A valid SC Report object

theKeyId

[in] Key Identifier for reporting data

theValue

[in] 32 bit Float value representation of the data

Remarks

The host or player can call this function to add either global, player-, or team-specific data. A report needs to be created before calling this function. For global keys, this function can only be called after starting global data. For player or teams, a new player or team needs to be added.

Section Reference: [Gamespy Competition SDK](#)

See Also: [scCreateReport](#), [scReportBeginGlobalData](#), [scReportBeginPlayerData](#), [scReportBeginTeamData](#), [scReportBeginNewPlayer](#), [scReportSetPlayerData](#), [scReportBeginNewTeam](#), [scReportSetTeamData](#)

scReportAddIntValue

Adds an integer value to the report for a specific key.

```
SCResult scReportAddIntValue(  
    SCReportPtr theReportData,  
    gsi_u16 theKeyId,  
    gsi_i32 theValue );
```

Routine	Required Header	Distribution
scReportAddIntValue	<sc.h>	SDKZIP

Return Value

Enum value used to indicate the specific result of the request. This will return `SCResult_NO_ERROR` if the request completed successfully.

Parameters

theReportData

[in] A valid SC Report object

theKeyId

[in] Key Identifier for reporting data

theValue

[in] 32 bit Integer value representation of the data

Remarks

The host or player can call this function to add either global, player-, or team-specific data. A report needs to be created before calling this function. For global keys, this function can only be called after starting global data. For player or teams, a new player or team needs to be added.

Section Reference: [Gamespy Competition SDK](#)

See Also: [scCreateReport](#), [scReportBeginGlobalData](#), [scReportBeginPlayerData](#), [scReportBeginTeamData](#), [scReportBeginNewPlayer](#), [scReportSetPlayerData](#), [scReportBeginNewTeam](#), [scReportSetTeamData](#)

scReportAddShortValue

Adds a short value to the report for a specific key.

```
SCResult scReportAddShortValue(  
    SCReportPtr theReportData,  
    gsi_u16 theKeyId,  
    gsi_i16 theValue );
```

Routine	Required Header	Distribution
scReportAddShortValue	<sc.h>	SDKZIP

Return Value

Enum value used to indicate the specific result of the request. This will return `SCResult_NO_ERROR` if the request completed successfully.

Parameters

theReportData

[in] A valid SC Report object

theKeyId

[in] Key Identifier for reporting data

theValue

[in] 16 bit Short value representation of the data

Remarks

The host or player can call this function to add either global, player-, or team-specific data. A report needs to be created before calling this function. For global keys, this function can only be called after starting global data. For player or teams, a new player or team needs to be added.

Section Reference: [Gamespy Competition SDK](#)

See Also: [scCreateReport](#), [scReportBeginGlobalData](#), [scReportBeginPlayerData](#), [scReportBeginTeamData](#), [scReportBeginNewPlayer](#), [scReportSetPlayerData](#), [scReportBeginNewTeam](#), [scReportSetTeamData](#)

scReportAddStringValue

Adds a string value to the report for a specific key.

```
SCResult scReportAddStringValue(  
    SCReportPtr theReportData,  
    gsi_u16 theKeyId,  
    const gsi_char * theValue );
```

Routine	Required Header	Distribution
scReportAddStringValue	<sc.h>	SDKZIP

Return Value

Enum value used to indicate the specific result of the request. This will return `SCResult_NO_ERROR` if the request completed successfully.

Parameters

theReportData

[in] A valid SC Report object

theKeyId

[in] The string key's identifier

theValue

[in] The string value

Remarks

The host or player can call this function to add either global, player-, or team-specific data. A report needs to be created before calling this function. For global keys, this function can only be called after starting global data. For player or teams, a new player or team needs to be added.

Section Reference: [Gamespy Competition SDK](#)

See Also: [scCreateReport](#), [scReportBeginGlobalData](#), [scReportBeginPlayerData](#), [scReportBeginTeamData](#), [scReportBeginNewPlayer](#), [scReportSetPlayerData](#), [scReportBeginNewTeam](#), [scReportSetTeamData](#)

scReportBeginGlobalData

Tells the competition SDK to start writing global data to the report.

```
SCResult scReportBeginGlobalData(  
    SCReportPtr theReportData );
```

Routine	Required Header	Distribution
scReportBeginGlobalData	<sc.h>	SDKZIP

Return Value

Enum value used to indicate the specific result of the request. This will return `SCResult_NO_ERROR` if the request completed successfully.

Parameters

theReportData

[ref] A valid SC Report Object

Remarks

After creating a report, this function should be called prior to writing global game data. Global data comes before player and team data. Note that keys and values can be recorded via the key/value utility functions.

Section Reference: [Gamespy Competition SDK](#)

See Also: [scCreateReport](#), [scReportAddIntValue](#), [scReportAddStringValue](#)

scReportBeginNewPlayer

Add a new player to the report.

```
SCResult scReportBeginNewPlayer(  
    SCReportPtr theReportData );
```

Routine	Required Header	Distribution
scReportBeginNewPlayer	<sc.h>	SDKZIP

Return Value

Enum value used to indicate the specific result of the request. This will return `SCResult_NO_ERROR` if the request completed successfully.

Parameters

theReportData

[in] A valid SC Report Object

Remarks

This function is used to be before adding new player data in the report. It tells the SDK that a new player needs to be added to the report.

Section Reference: [Gamespy Competition SDK](#)

See Also: [scCreateReport](#), [scReportBeginPlayerData](#), [scReportSetPlayerData](#), [scReportAddIntValue](#), [scReportAddStringValue](#)

scReportBeginNewTeam

Adds a new team to the report.

```
SCResult scReportBeginNewTeam(  
    SCReportPtr theReportData );
```

Routine	Required Header	Distribution
scReportBeginNewTeam	<sc.h>	SDKZIP

Return Value

Enum value used to indicate the specific result of the request. This will return `SCResult_NO_ERROR` if the request completed successfully.

Parameters

theReportData

[in] A valid SC Report Object

Remarks

After the beginning of any team data is set, this function can be called to start a new team. After this function has been called, the game can start adding team data to the report.

Section Reference: [Gamespy Competition SDK](#)

See Also: [scCreateReport](#), [scReportBeginTeamData](#), [scReportSetPlayerData](#), [scReportAddIntValue](#), [scReportAddStringValue](#)

scReportBeginPlayerData

Tells the competition SDK to start writing player data to the report.

```
SCResult scReportBeginPlayerData(  
    SCReportPtr theReportData );
```

Routine	Required Header	Distribution
scReportBeginPlayerData	<sc.h>	SDKZIP

Return Value

Enum value used to indicate the specific result of the request. This will return `SCResult_NO_ERROR` if the request completed successfully.

Parameters

theReportData

[in] A valid SC Report Object

Remarks

Use this function to mark the starting of player data. Player data should come after global data, and before team data. The game can start adding each player and its specific data after this is called.

Section Reference: [Gamespy Competition SDK](#)

See Also: [scCreateReport](#), [scReportBeginNewPlayer](#), [scReportSetPlayerData](#), [scReportAddIntValue](#), [scReportAddStringValue](#)

scReportBeginTeamData

Tells the competition SDK to start writing player data to the report.

```
SCResult scReportBeginTeamData(  
    SCReportPtr theReportData );
```

Routine	Required Header	Distribution
scReportBeginTeamData	<sc.h>	SDKZIP

Return Value

Enum value used to indicate the specific result of the request. This will return `SCResult_NO_ERROR` if the request completed successfully.

Parameters

theReportData

[in] A valid SC Report Object

Remarks

Use this function to mark the starting of team data. Team data should come after global data, and player data. The game can start adding each team and its specific data after this is called.

Section Reference: [Gamespy Competition SDK](#)

See Also: [scCreateReport](#), [scReportBeginNewTeam](#), [scReportSetTeamData](#), [scReportAddIntValue](#), [scReportAddStringValue](#)

scReportEnd

Denotes the end of a report for the report specified.

```
SCResult scReportEnd(  
    SCReportPtr theReport,  
    gsi_bool isAuth,  
    SCGameStatus theStatus );
```

Routine	Required Header	Distribution
scReportEnd	<sc.h>	SDKZIP

Return Value

Enum value used to indicate the specific result of the request. This will return `SCResult_NO_ERROR` if the request completed successfully.

Parameters

theReport

[in] A valid SC Report Object

isAuth

[in] Authoritative report

theStatus

[in] Final Status of the reported game

Remarks

Used to set the end of a report. The report must have been properly created and have some data. Any report being submitted requires that function be called before the submission. Incomplete reports will be discarded.

Section Reference: [Gamespy Competition SDK](#)

See Also: [scCreateReport](#), [scSubmitReport](#), [SCGameStatus](#)

scReportSetAsMatchless

Called after creating the report to set it as a matchless report - this is needed if the report is being submitted to a "matchless" game session.

```
SCResult scReportSetAsMatchless(  
    SCReportPtr theReport );
```

Routine	Required Header	Distribution
scReportSetAsMatchless	<sc.h>	SDKZIP

Return Value

Enum value used to indicate the specific result of the request. This will return `SCResult_NO_ERROR` if the request completed successfully.

Parameters

theReport

[ref] A valid SC Report Object

Remarks

This should not be used for a non-matchless session report.

Section Reference: [Gamespy Competition SDK](#)

See Also: [scCreateMatchlessSession](#), [scCreateReport](#)

scReportSetPlayerData

Sets initial player data in the report specified.

```
SCResult scReportSetPlayerData(  
    SCReportPtr theReport,  
    gsi_u32 thePlayerIndex,  
    const gsi_u8  
    thePlayerConnectionId[SC_CONNECTIONID_LENGTH],  
    gsi_u32 thePlayerTeamIndex,  
    SCGameResult theResult,  
    gsi_u32 theProfileId,  
    const GSLoginCertificate * theCertificate,  
    const gsi_u8 theAuthData[16] );
```

Routine	Required Header	Distribution
scReportSetPlayerData	<sc.h>	SDKZIP

Return Value

Enum value used to indicate the specific result of the request. This will return `SCResult_NO_ERROR` if the request completed successfully.

Parameters

theReport

[ref] A valid SC Report Object

thePlayerIndex

[in] Index of the player (0 - Number of players)

thePlayerConnectionId

[in] Connection ID that the player received from the competition backend

thePlayerTeamIndex

[in] Team index of the player, if that player is on a team.

theResult

[in] Standard SC Game result

theProfileId

[in] Profile ID of the player

theCertificate

[in] Certificate obtained from the auth service.

theAuthData

[in] Authentication data

Remarks

A report must have been created prior to using this function. Each player must have a valid login certificate from the authentication service also. This function should be called after a new player is added to the report. Any key/value pairs that need to be added should be done after calling this function.

Section Reference: [Gamespy Competition SDK](#)

See Also: [scCreateReport](#), [scReportBeginPlayerData](#), [scReportBeginNewPlayer](#), [scReportAddIntValue](#), [scReportAddStringValue](#)

scReportSetTeamData

Sets the initial team data in the report specified.

```
SCResult scReportSetTeamData(  
    SCReportPtr theReport,  
    gsi_u32 theTeamIndex,  
    SCGameResult theResult );
```

Routine	Required Header	Distribution
scReportSetTeamData	<sc.h>	SDKZIP

Return Value

Enum value used to indicate the specific result of the request. This will return `SCResult_NO_ERROR` if the request completed successfully.

Parameters

theReport

[in] A valid SC Report Object

theTeamIndex

[in] The index of the team being reported

theResult

[in] The team's result (e.g. win, loss, draw)

Remarks

A report must have been created prior to using this function. This function should be called after a new team is added to the report. Any key/value pairs that need to be added should be done after calling this function.

Section Reference: [Gamespy Competition SDK](#)

See Also: [scCreateReport](#), [scReportBeginTeamData](#), [scReportBeginNewTeam](#), [scReportAddIntValue](#), [scReportAddStringValue](#)

scSetReportIntention

Called to tell the backend the type of report that the player or host will send.

```
SCResult scSetReportIntention(  
    const SCInterfacePtr theInterface,  
    const gsi_u8 theConnectionId[SC_CONNECTION_GUID_SIZE],  
    gsi_bool isAuthoritative,  
    const GSLoginCertificate * theCertificate,  
    const GSLoginPrivateData * thePrivateData,  
    SCSetReportIntentionCallback theCallback,  
    gsi_time theTimeoutMs,  
    const void * theUserData );
```

Routine	Required Header	Distribution
scSetReportIntention	<sc.h>	SDKZIP

Return Value

Enum value used to indicate the specific result of the request. This will return `SCResult_NO_ERROR` if the request completed successfully.

Parameters

theInterface

[ref] A valid SC Interface Object.

theConnectionId

[in] The player's former ConnectionId if he was previously in the same match. Set to NULL if unused.

isAuthoritative

[in] flag set if the snapshot being reported will be an authoritative.

theCertificate

[ref] Certificate obtained from the authentication web service.

thePrivateData

[ref] Private data obtained from the authentication web service.

theCallback

[ref] The callback called when set report intention completes.

theTimeoutMs

[in] The amount of time to spend on the operation before a timeout occurs.

theUserData

[ref] Application data that may be used in the callback.

Remarks

The should be called by both the host and client before sending a report. The host should have created a session before calling this. It allows the server to know ahead of time what type of report will be sent. Reports submitted without an intention will be discarded.

Section Reference: [Gamespy Competition SDK](#)

See Also: [scCreateSession](#), [SCSetReportIntentionCallback](#), [scSubmitReport](#)

scSetSessionId

Used to set the session ID for the current game session.

```
SCResult scSetSessionId(  
    const SCInterfacePtr theInterface,  
    const gsi_u8 theSessionId[SC_SESSION_GUID_SIZE]);
```

Routine	Required Header	Distribution
scSetSessionId	<sc.h>	SDKZIP

Return Value

Enum value used to indicate the specific result of the request. This will return `SCResult_NO_ERROR` if the request completed successfully.

Parameters

theInterface

[in] A valid SC Interface Object

theSessionId

[in] The session ID - this has a constant length of
SC_SESSION_GUID_SIZE

Remarks

The session ID identifies a single game session happening between players. Players should use the `scGetSessionId` function in order to obtain the session ID. This should not be called if a session has not yet been created.

Section Reference: [Gamespy Competition SDK](#)

See Also: [scGetSessionId](#), [scCreateSession](#)

scShutdown

Shuts down the Competition SDK.

```
SCResult scShutdown(  
    SCInterfacePtr theInterface );
```

Routine	Required Header	Distribution
scShutdown	<sc.h>	SDKZIP

Return Value

Enum value used to indicate the specific result of the request. This will return `SCResult_NO_ERROR` if the request completed successfully.

Parameters

theInterface

[in] A valid SC Inteface Object

Remarks

In order to clean up all resources used by the SDK, this interface function must be called. Do not call this function if you plan to continue reporting stats.

Section Reference: [Gamespy Competition SDK](#)

See Also: [sclInitialize](#)

scSubmitReport

Initiates the submission of a report.

```
SCResult scSubmitReport(  
    const SCInterfacePtr theInterface,  
    const SCReportPtr theReport,  
    gsi_bool isAuthoritative,  
    const GSLoginCertificate * theCertificate,  
    const GSLoginPrivateData * thePrivateData,  
    SCSubmitReportCallback theCallback,  
    gsi_time theTimeoutMs,  
    void * theUserData );
```

Routine	Required Header	Distribution
scSubmitReport	<sc.h>	SDKZIP

Return Value

Enum value used to indicate the specific result of the request. This will return `SCResult_NO_ERROR` if the request completed successfully.

Parameters

theInterface

[in] A valid SC Interface Object.

theReport

[in] A valid SC Report object

isAuthoritative

[in] Flag to tell if the snapshot is authoritative

theCertificate

[in] Certificate Obtained from the auth service.

thePrivateData

[in] Private Data Obtained from the auth service.

theCallback

[in] Callback to be called when submit report completes.

theTimeoutMs

[in] The amount of time before a timeout occurs

theUserData

[in] Application data that may be used in the callback

Remarks

Once the report has been completed with a call to `scReportEnd`, the player or host can call this function to submit a report. The certificate and private data are both required to submit a report. Incomplete reports will be discarded. The callback passed in will tell the game the result of the operation.

Section Reference: [Gamespy Competition SDK](#)

See Also: [scInitialize](#), [scCreateSession](#), [scSetReportIntention](#), [scReportEnd](#), [SCSubmitReportCallback](#)

scThink

Called to complete pending operations for functions with callbacks.

```
SCResult scThink(  
    SCInterfacePtr theInterface );
```

Routine	Required Header	Distribution
scThink	<sc.h>	SDKZIP

Return Value

Enum value used to indicate the specific result of the request. This will return `SCResult_NO_ERROR` if the request completed successfully.

Parameters

theInterface

[in] A valid SC Inteface Object

Remarks

This function should be called with a valid interface object. It will take care of pending requests that have been made by the interface functions.

Section Reference: [Gamespy Competition SDK](#)

See Also: [scInitialize](#), [scCreateSession](#), [scSetReportIntention](#), [scSubmitReport](#)

Competition SDK Callbacks

[SCCreateSessionCallback](#)

Called when scCreateSession has completed.

[SCSetReportIntentionCallback](#)

Called when scReportIntention has completed.

[SCSubmitReportCallback](#)

Called when scSubmitReport completes.

SCCreateSessionCallback

Called when scCreateSession has completed.

```
typedef void (*SCCreateSessionCallback)(  
    const SCInterfacePtr theInterface,  
    GHTTPResult theHttpRequest,  
    SCResult theResult,  
    const void * theUserData );
```

Routine	Required Header	Distribution
SCCreateSessionCallback	<sc.h>	SDKZIP

Parameters

theInterface

[in] the pointer to the SC Interface object. The game usually has copy of this also.

theHttpRequest

[in] Http result from creating a session

theResult

[in] SC Result telling the application what happened when creating a session

theUserData

[in] constant pointer to user data

Remarks

Called when a game session is created. The results will determine if the session was successfully created. If there were any errors, theResult will be set to the specific error code. Otherwise theResult will be set to SCResult_NO_ERROR. Please see SCResult for error codes.

Section Reference: [Gamespy Competition SDK](#)

See Also: [scCreateSession](#), [SCResult](#)

SCSetReportIntentionCallback

Called when scReportIntention has completed.

```
typedef void (*SCSetReportIntentionCallback)(  
    const SCInterfacePtr theInterface,  
    GHTTPResult theHttpRequest,  
    SCResult theResult,  
    const void * theUserData );
```

Routine	Required Header	Distribution
SCSetReportIntentionCallback	<sc.h>	SDKZIP

Parameters

theInterface

[ref] the pointer to the SC Interface object. The game usually has copy of this also.

theHttpRequest

[in] Http result from creating a session

theResult

[in] SC Result telling the application what happened when creating a session

theUserData

[ref] constant pointer to user data

Remarks

Called when a host or client reporting its intention is complete. The results will determine if the session was successfully created. If there were any errors, theResult will be set to the specific error code. Otherwise theResult will be set to SCResult_NO_ERROR. Please see SCResult for error codes.

Section Reference: [Gamespy Competition SDK](#)

See Also: [scSetReportIntention](#), [SCResult](#)

SCSubmitReportCallback

Called when scSubmitReport completes.

```
typedef void (*SCSubmitReportCallback)(  
    const SCInterfacePtr theInterface,  
    GHTTPResult theHttpRequest,  
    SCResult theResult,  
    const void * theUserData );
```

Routine	Required Header	Distribution
SCSubmitReportCallback	<sc.h>	SDKZIP

Parameters

theInterface

[in] the pointer to the SC Interface object. The game usually has copy of this also.

theHttpRequest

[in] Http result from creating a session

theResult

[in] SC Result telling the application what happened when creating a session

theUserData

[in] constant pointer to user data

Remarks

After the SDK submits the report, the backend will send back results that will be available in this callback. If there were any errors, theResult will be set to the specific error code. Otherwise theResult will be set to SCResult_NO_ERROR. Please see SCResult for error codes.

Section Reference: [Gamespy Competition SDK](#)

See Also: [scSubmitReport](#), [SCResult](#)

Competition SDK Enumerations

[SCGameResult](#)

Used when submitting a report for a game session to reflect the player's result.

[SCGameStatus](#)

The Game Status Indicates how the session ended and is declared when ending a report.

[SCResult](#)

used for checking errors and failures

SCGameResult

Used when submitting a report for a game session to reflect the player's result.

```
typedef enum  
{  
    SCGameResult_WIN,  
    SCGameResult_LOSS,  
    SCGameResult_DRAW,  
    SCGameResult_DISCONNECT,  
    SCGameResult_DESYNC,  
    SCGameResult_NONE,  
    SCGameResultMax  
} SCGameResult;
```

Constants

SCGameResultMax

Total number of game result codes.

Remarks

Can be used for both player and a team.

Section Reference: [Gamespy Competition SDK](#)

SCGameStatus

The Game Status Indicates how the session ended and is declared when ending a report.

```
typedef enum  
{  
    SCGameStatus_COMPLETE,  
    SCGameStatus_PARTIAL,  
    SCGameStatus_BROKEN,  
    SCGameStatusMax  
} SCGameStatus;
```

Constants

SCGameStatus_COMPLETE

The game session came to the expected end without interruption (disconnects, desyncs). This status indicates that game results are available for all players.

SCGameStatus_PARTIAL

Although the game session came to the expected end, one or more players unexpectedly quit or were disconnected. Game results should explicitly report which players were disconnected to be used during normalization for possible penalty metrics.

SCGameStatus_BROKEN

The game session did not reach the expected end point and is incomplete. This should be reported when there has been an event detected that makes the end result indeterminate.

SCGameStatusMax

Total number of game status codes.

Remarks

For **SCGameStatus** reporting, the game should do the following. As long as the game finished properly, and no one disconnected during the course of play, then all players in the match should submit **SCGameStatus_COMPLETE** reports. If any members disconnected during play, but the game was finished completely, then all players in the match should submit **SCGameStatus_PARTIAL** reports indicating that disconnects occurred. For any players who do not complete the match, a **SCGameStatus_BROKEN** report should be submitted. Thus if the game did not completely finish, all players will submit broken reports. The only case that will trigger an invalid report is if reports for the same game describe status as both **SCGameStatus_COMPLETE** and **SCGameStatus_PARTIAL**. Since COMPLETE indicates that all players finished the game w/o a disconnect and PARTIAL indicates that disconnects occurred, at no time should a game report both complete and partial - this will be seen as an exploit and invalidate the report.

Section Reference: [Gamespy Competition SDK](#)

SCResult

used for checking errors and failures.

```
typedef enum  
{  
    SCResult_NO_ERROR= 0,  
    SCResult_NO_AVAILABILITY_CHECK,  
    SCResult_INVALID_PARAMETERS,  
    SCResult_NOT_INITIALIZED,  
    SCResult_CORE_NOT_INITIALIZED,  
    SCResult_OUT_OF_MEMORY,  
    SCResult_CALLBACK_PENDING,  
    SCResult_HTTP_ERROR,  
    SCResult_UNKNOWN_RESPONSE,  
    SCResult_RESPONSE_INVALID,  
    SCResult_REPORT_INCOMPLETE,  
    SCResult_REPORT_INVALID,  
    SCResult_SUBMISSION_FAILED,  
    SCResult_UNKNOWN_ERROR,  
    SCResultMax  
} SCResult;
```

Constants

SCResult_NO_ERROR

No error has occurred.

SCResult_NO_AVAILABILITY_CHECK

The standard GameSpy Availability Check was not performed prior to initialization.

SCResult_INVALID_PARAMETERS

Parameters passed to interface function were invalid.

SCResult_NOT_INITIALIZED

The SDK was not initialized.

SCResult_CORE_NOT_INITIALIZED

The core was initialized by the application.

SCResult_OUT_OF_MEMORY

The SDK could not allocate memory for its resources.

SCResult_CALLBACK_PENDING

Result tell the application, that the operation is still pending.

SCResult_HTTP_ERROR

Error occurs if the backend fails to respond with correct HTTP.

SCResult_UNKNOWN_RESPONSE

Error occurs if the SDK cannot understand the result.

SCResult_RESPONSE_INVALID

Error occurs if the SDK cannot read the response from the backend.

SCResult_REPORT_INCOMPLETE

The report was incomplete.

SCResult_REPORT_INVALID

Part or all of report is invalid.

SCResult_SUBMISSION_FAILED

Submission of report failed.

SCResult_UNKNOWN_ERROR

Error unknown to sdk.

SCResultMax

Total number of result codes that can be returned.

Remarks

Results of a call to an interface function or operation. It can be used to see if the initial call to a function completed without error. The callback that is passed to interface functions will also have a value that is of this type. The application can check this value for failures.

Section Reference: [Gamespy Competition SDK](#)

CD Key SDK

Overview

The GameSpy CDKey SDK is a simple toolkit designed to allow developers to add secure, server-based CD Key validation to their games. Server-based CD Key validation has proven to be the only widely successful method of combating piracy available today.

In server-based CD Key validation, a client sends its CD Key to the game server / host when it wants to join a multiplayer game. The server checks with a validation server on the backend to make sure that the CD Key is valid. If it isn't, the server refuses the connection. The validation server also ensures that no two players can use the same key at the same time. Of course the key is always encoded so that neither the server operator nor someone "sniffing" the connection can steal the CD Key.

Several other common anti-piracy methods are:

Client-based CD Keys / Serial numbers

Can easily be "cracked" and removed

CD Check / CD anti-copy measures

As long as the data on the disc can be read, it can be copied and the CD Check can be cracked

Overburn / 80 minute CDs

Recordable 80 minute CDs are now widely available, and CD emulators can often get around this protection as well

Server-based CD Key validation works because it is controlled completely by the server/host - a client cannot "crack" any part of their local code to give the correct response to the server without a valid CD Key. While server-based CD Key validation is not "perfect", any flaws that exist are in the implementation, not the concept.

Please note that server-based CD Key validation is only appropriate for a certain class of games. While the CD Keys can also be checked on the client side to help protect the single player game, they cannot add any

greater amount of protection to the single player (non-Internet) portion of the game than a normal CD key check would. The only thing that server-based CD Key validation can do 100% effectively is prevent clients without valid keys from playing on public Internet servers. However, for games that are primarily multi-player, or have a large multi-player component, this can be a large deterrent to piracy (both large scale "bootlegs" and small-scale "sharing").

We also feel it is important to point out that there are many "production" problems that can occur with CD Keys of any sort, and can potentially impact both the effectiveness of the protection and the number of support issues that come up.

Some common problem with CD Keys include:

- Labeling errors during duplication / packaging leading to incorrect or missing CD Keys
- Users mistyping the CD Key
- Users "losing" their CD Key (especially if they need to reinstall it on a new machine)
- Users "sharing" their CD Key without being aware of the consequences (i.e. they won't be able to play online any more)

Less common problems include:

- An internal "leak" of the valid CD Key list which ends up on the Internet
- Users buying the software, getting the CD Key, and then returning it

Once these problems are overcome or accepted, server-based CD Key validation offers some unique features not present in any other anti-piracy scheme. These features include:

- The option to delay enforcement of the protection until the game has generated "critical mass"
- The ability to actually track usage of pirated vs. legal copies of the game
- Hourly and daily numbers for tracking play of the game online

- Unique potential for data mining

All of the data that is tracked is done completely anonymously so that the privacy of your users is protected.

There are actually three layers of protection provided by server-based CD Key validation, each layer targeted at stopping a particular type of piracy:

- You must have a valid CD Key to play online
- No two people can play online at the same time with the same CD Key
- Any valid CD Key can be disabled if it is distributed / abused

This CD Key SDK consists of two very simple portable C APIs - one for the client that encodes the CD Key for sending to the server and another for the server that sends the CD Key to the validation server and authenticates the clients. We believe all of the code to be disclosure safe - in other words, even if the entire source for the system were published, it would be impossible to circumvent it. This is one of the reasons we feel confident in distributing full-source to developers - we welcome your attempts to "break" the system, even from within. We still obfuscate some of the communications to help guard against the "annoyance" factor of thousands of hackers trying to break into our key server through a plain-text interface, but even without this, the system would be totally secure.

Additions to your current code will be fairly minimal, and there is plenty of flexibility for you to implement the SDK in a way best suited to your game.

Fully working examples of both the server and client code are included for testing / reference. This document provides a step by step set of instructions for implementing the CD Key SDK.

How It Works

Terms

The following terms are used throughout this document.

Server

The machine that is "hosting" the game and to which the clients connect

Host

Same as a server

Client

A single player / machine that connects to a server / host

User

Same as a client

Validation Server

The server run by GameSpy which validates CD Keys and tracks online users

Process

1. A list of valid CD Keys is generated by the developer and put on the CD cases during packing. The keys must be self-validating, i.e. there is a function that can determine whether the key is mathematically valid. An example of a CD Key generation/validation pair is included in the SDK. The actual CD Keys used should be less than a 0.0001% subset of the possible keys (to assure that "guessing" a valid key is nearly impossible)
2. On install/run of the client, the user inputs the CD Key. The client validates that the CD Key is (mathematically) correct to check typos / made up keys and allows the user to play the game.
3. On connection to a server, a handshake occurs to exchange the key
 - a. The server sends a "challenge" string of random data to the client - *note that this challenge string can contain a maximum of 32 characters.*

- b. The client computes a set of hashes based on the challenge, its CD key, and a random value and passes them back to the server
- c. The server sends the challenge and hashes to the validation server
- d. The validation server checks its CD Key database to determine whether the hashes are valid. If they are not, it returns an error to the server.
- e. The validation server then checks to see if another user with that CD Key is online. If one is, then it first queries the old server, to make sure that user is still connected (in case the server crashed), and if they are, it returns an error to the new server.
- f. When the client logs off or the server shuts down a message is sent to the validation server to take the CD key offline.

Miscellaneous

All game server to validation server messaging is done via UDP, and in case of a dropped packet or missing data, a "positive" result is always assumed (so no user with a valid key will EVER be locked out). Because the protocol uses UDP, there is always a chance that the validation or reply packet might get dropped, allowing a user with an invalid CD Key to play, but the chances of this occurring is quite small (probably 1-2% or less for most servers).

It is technically possible for a cracker to modify the server code to prevent it from checking CD Keys (i.e. allow any user to connect / play on that server). This does not tend to be an issue in most cases, since the vast majority of server operators / game hosts will want to prevent pirates from playing on their servers. However, we have specifically designed the code to be difficult to find and disable on the server. For additional protection, you can choose to allow the CDKey SDK integrate with the Query and Reporting 2 SDK. If a cracker attempts to prevent the server from validating CDKeys, they will end up preventing the server from being listed on the public server list. Of course the most important fact is that there is nothing that can be done on the game client-side to remove or weaken the CDKey protection.

Internet-based validation obviously does not apply to games played on a local LAN (not connected to the net), and in general, the code does not need any changes to reflect this (since no reply will be returned from the validation server, all clients will be considered valid). However, the current server API does do a local check of CD Keys, so that no two players with the same key can connect to the same server (even on a local LAN). You may wish to change this functionality to allow 2 or more players to "share" a CD Key on a local LAN (e.g. for "clone" installs).

Testing

When you are ready to begin testing your implementation of the SDK, you can start by using game ID "0" and the test keys listed below.

Once you've generated your own list of unique keys for your game, you can use the web administration interface described below to add them. The list of keys can be changed or added to later if needed.

Test Keys

2dd4-893a-ce85-6411
4bdb-27e9-ecf8-c042
6585-2eeb-c544-9dd2
42ea-082e-74e5-15b6
7bca-b5e2-47e4-42d1
47a0-84e7-bf51-16f4
899e-040f-fc85-72eb
1156-ba66-a3f2-47b3
22f2-dce2-ce67-c8aa
9131-3dd3-ceb6-c292
5022-bcea-5312-4348
468b-bb7e-f5f8-3936

Web Administration Interface

The CDKey SDK is supported by a full-featured web interface for administering individual keys, batches of keys, and obtaining usage and abuse reports. Multiple users from the publisher and developer can be set up with accounts for secure access to the site.

Account Setup

Each user that requires access to the system will need to have an account set up with specific permissions. To set up an account, [contact devsupport@gamespy.com](mailto:devsupport@gamespy.com).

You will need to specify the e-mail address of the user who needs access, and the set of permissions they will be granted.

The following individual permissions are available:

View key reports

Allows the user to view any of the pages in the reports section (detailed below)

Add new keys

Allows the user to add individual keys or batches of keys for the game. Note that you will be billed for any keys added in accordance with your licensing agreement.

Enable / Disable keys

Allows the user to enable or disable single keys or batches of keys

View key list

Allows the user to download a plain-text listing of all keys for in a batch. This should only be used for testing and key list verification.

Requests to disable accounts should also be sent to devsupport@gamespy.com.

Authentication is done via the GameSpyID system. All users should sign up for a GameSpyID at www.gamespyid.com prior to contacting the developer relations staff for account setup.

Once the account has been set up, the developer relations team will contact the user with the appropriate URL for accessing the admin system.

Main Menu

After logging in to the CDKey Admin site, a menu of options will be available, based on the permissions granted to the active account. Each option is described in detail below.

View Key List

Selecting this option will allow you to select a batch of keys and download them in a plaintext file, one key per line. This can be used to verify the list of keys for a batch against other sources. Users must have the "View key list" permission to access this page.

Enable / Disable Keys

This page allows an admin to enable or disable single keys or batches of keys. On the top portion of the page is a list of key batches. Uncheck a batch to disable the entire batch, or check it to enable. This can be used to disable beta keys or press keys after they should no longer be used (assuming those keys have been added as a separate batch).

The page also includes a text entry box where you can paste a list of individual keys, one per line, to be disabled or enabled. Typically this is used to disable keys from returned copies of games or when a known set of keys needs to be disabled. The enable/disable keys permission is required to use this page.

Add New Keys

Keys are added to the CDKey system in batches. Each batch has 1 or more keys in it (typically thousands) and can have a name and comment associated with it. For example, if you generate a separate list of keys for each region your game will ship in, you can upload them and name them individually. This allows you better control if the keys for one region or

pressing of your game are destroyed, and allows you to view some reports broken up by the batch a key was in.

On the Add New Keys page you should specify a batch name and comment, then select the keys to add. You can either upload an ASCII list of keys from your hard drive (one key per line), or if you are adding only a small number of keys, you can paste them into the provided text box.

Keys can also be generated by providing a set of generation parameters.

A user must have the Add new keys permission to add a new batch of keys.

Reports

The CDKey system includes a number of reports that will help keep you informed about what is happening with your game online, research key usage, and detect abused keys so they can be disabled.

Users must have the View key reports permission to view these reports.

Batch Information

The batch information report provides a summary of information about the individual batches of keys that have been uploaded for you game. Each batch has the following information provided for it:

Batch name

The name of the batch, provided when it was uploaded

Add date

The date the batch was first added to the system

Admin account

GameSpyID number of the admin who added the batch. You can click this link to see the user details.

Comment

Comment that was provided when the batch was created (if any)

Disabled

Flag that shows whether the batch has been disabled

Total keys

Total number of keys for this batch

Total used keys

Number of distinct keys from the batch that have been used online at least once

Total active keys

Number of distinct keys from the batch that have been used in the last 30 days

A totals line is provide that sums the numbers from all batches.

Overall Usage

The overall usage report gives usage information for all batches of keys between a range of dates. To generate the report, you select the start date, end date, and data interval you are interested in.

Reports can be generated by hour, day, week, or month.

Each interval on the date range specified will contain the following data:

- Total number of authentication attempts on that date (valid or invalid)
- Number of authentications that were denied due to an invalid CDKey
- Number of authentications that failed due to the CDKey already being online
- Number of keys that were authenticated for the first time on that date (i.e. "new users")
- Number of authentication attempts for disabled CDKeys (may include multiple for the same disabled key if multiple attempts were made)

Key Information

The key information report allows you to specify a specific CDKey to get information about it and its usage history. This report is also linked from other reports to give additional information about a specific key. If the user has Enable/Disable key rights, a button is available on this page to disable the key (or re-enable it).

The following information is available about each key:

Enabled

Flag that indicates whether the key has been disabled

Origin Batch

Name of the CDKey batch that includes this key

Plain Key

The plain-text value of the key

Key Hash Value

The hashed version of the key (which is what is sent from client to server)

Total Uses

The total number of successful authentications with this key

Total Conflicts

Total number of times someone attempted to use the key while it was already in use

Recent Conflicts

Number of times the key was in conflict during a recent period of time (currently 7 days).

Recent Use History

This list has the last 20 dates/times and client IP addresses that used the key. It can help determine whether a key is being shared by multiple users.

Abuse History

This list shows the last 100 instances of abuse for the key, including a reason for the abuse (e.g. the key was already online), and the client/server IPs that were involved in the conflict.

Disabled Keys

The disabled key report is used to view the list of recently disabled CDKeys. The date and user that disabled the key are available, and a link is provided to the key to view key information and re-enable the key if desired.

Abuse Information

The Abuse Information report allows you to view the top keys in conflict (e.g. someone attempted to use the key multiple times online at the same time) on a specific range of dates. For example, you can set it for the past week to see just the keys that were abused in the past week. The keys are sorted from most abused to least, and a link is provided to the key information page for the key, where the abuse can be investigated and the key can be disabled.

File Manifest

The following files should be included with this package. If any of the files are missing, please [contact devsupport@gamespy.com](mailto:contact_devsupport@gamespy.com).

File	Description
<i>gcdkeyc.c</i>	Client API code
<i>gcdkeyc.h</i>	Client API Header file
<i>md5c.c</i>	MD5 Hash code
<i>md5.h</i>	MD5 Hash Header
<i>gcdkeyclienttest.c</i> server application	Sample client application, talks to the sample server application
<i>gcdkeyclient.dsp</i>	DevStudio project for the Client API / sample
<i>gcdkeys.c</i>	Server API code
<i>gcdkeys.h</i>	Server API Header file
<i>nonport.c</i>	System-dependant code (sockets, etc)
<i>nonport.h</i>	Header for system-dependant code
<i>gcdskeyservertest.c</i> from sample client	Sample server application, accepts connections from sample client
<i>gcdkeyserver.dsp</i>	DevStudio project for Server API / sample
<i>gcdkeygen.c</i>	Sample key generation / validation code
<i>gcdkeygen.dsp</i>	DevStudio project for key gen sample
<i>gcdkey.dsw</i> keygen projects	DevStudio workspace with client, server, and keygen projects
<i>gcdkeyserver_qr2.dsp</i>	DevStudio project for Server API / QR2 Integration sample
<i>gcdkeyservertest_qr2.c</i>	Sample server application, plus integration with QR2 SDK

Implementation

Step 0: (Server) Initialize the CD Key API, Think, and Shutdown

Somewhere in your server startup code, call `gcd_init` with the game ID you have been given to initialize the API sockets and structures. The SDK supports using multiple game IDs simultaneously. You may need to do this if, for example, you need to authenticate multiple CDKeys per-use (e.g. one for the main game, one for a mission pack), or your game server supports multiple products and needs to authenticate each product separately.

In your main game / message loop, call `gcd_think` to allow the API to process any pending authorization requests / messages. This function should be called at least once every 10-100ms and is guaranteed not to block (although it may make a callback if an authorization response has come in). If your game uses the Query and Reporting 2 SDK, you can place this call in the same area as the call to `qr2_think`.

In your server shutdown code call `gcd_shutdown` to release the socket and send disconnect messages to the validation server for any clients still on the server.

Step 1: (Server) Send a challenge string to the client

During the client connection process you need to send the client a random challenge string. This challenge will be used as part of the response hash. You will need to pass this challenge along with the user's response to the `gcd_authenticate_user` function, so be sure to hold onto it. The challenge string can be any combination of letters / digits. 6-8 characters should be adequate, and the string has a maximum limit of 32 characters.

Step 2: (Client) Respond to the challenge

When the client receives the challenge string it should calculate a response using the `gcd_compute_response` function in the Client API.

Pass the client's CD key and the challenge string into the function and it

will return the response string, a 72 character ASCII string. Send this response back to the server.

Step 3: (Server) Begin the authentication process

Once you have received the client's response, you can call `gcd_authenticate_user` to send an authentication request.

```
void gcd_authenticate_user(int gameid, int localid,
    unsigned int userip, char *challenge, char *response,
    AuthCallbackFn authfn, RefreshAuthCallbackFn
    refreshfn, void *instance);
```

`gameid`

the game ID issued for your game

`localid`

a unique int used to identify each client on the server. No two clients should have the same localid.

`userip`

is the client's IP address, preferably in network byte order

`challenge`

the challenge string that was sent to the client

`response`

the response that the client received

`authfn`

a callback that is called when the user is either authorized or rejected. This function will be called within two seconds of `gcd_authenticate_user`, even if the validation server hasn't responded yet.

`instance`

any user-defined data you want to pass into the callback function (e.g. an object or structure pointer, or NULL). The example server uses this to pass in the array of client structures.

This function will return immediately, and you will have to wait until the callback is triggered to determine whether the client is valid or not. During

this period (usually 100ms or less, but up to 2 sec max) you can hold the client in a limbo-state, or allow them to enter the game (and disconnect them if a negative response comes back).

Remember that you need to be calling `gcd_think` during this time, or the callback will never be triggered. You should be calling `gcd_think` even when not waiting for a callback, since it also handles processing on "online" queries from the validation server.

Step 4: (Server) Create the Callback

You will need to create a callback function that is called once the validation server responds with the client's authorization status (or a 2 second timeout occurs).

The prototype for this function is:

```
void AuthCallbackFn(int gameid, int localid, int
authenticated, char *errmsg, void *instance);
```

`gameid`

the game ID you requested authentication for

`localid`

the id that you passed into `gcd_authenticate_user`, and indicates which user this callback is referring to (since multiple authentication requests can be sent before the first is returned).

`authenticated`

a 1/0 value that indicates whether the user was authenticated or not.

If the user was not authenticated, `errmsg` contains a descriptive string of the reason (either CD Key not valid, or CD Key in use). `Errmsg` is never `NULL`, so if there is no message it will be an empty string.

`instance`

the user-defined data that you requested be passed to the callback

If the client was authenticated you should allow them to continue / enter

the game. If not, you should send an error message to the client and disconnect / disable them. You do not need to call `gcd_disconnect_user` (but you can) as they have already been removed from the APIs internal structures.

Step 5: (Server) Create the reauth Callback

The server should have the reauth callback function defined for reauthentications:

The prototype for this function is:

```
void RefreshAuthCallbackFn(int gameid, int localid,
int hint, char *challenge, void *instance);
```

`gameid`

the the game id used to initialize the SDK with

`localid`

the index of the player

`hint`

a session id for a client used for reauthentication - this is the key passed into `gcd_process_reauth`

`challenge`

a challenge string used for reautentication

`instance`

the user-defined data that you requested be passed to the callback

This function will be called when the validation server requires proof that a player is still online using the cd key being checked. The server needs to send the challenge to the player via its own socket. The player must call `gcd_compute_response` in order to create a new response. The host/server in turn uses this response to call `gcd_process_reauth` so that it can prove the client's existence. Otherwise the validation server will consider that client offline.

Step 6: (Server) Call Disconnect when a user leaves

When a user disconnects / logs off the server you should call `gcd_disconnect_user` immediately so that the validation server can be notified that the user is now offline and the CD Key is marked as available again. If you fail to call `gcd_disconnect_user`, the user may, in some cases, have trouble connecting to another server (since the validation server AND your game server both think the user is still playing).

Don't be concerned about no notification being sent in the case of server crashes / sudden shutdowns - a user will only be denied access if your server is still responding and thinks the user is online. Any time a "conflict" occurs (a user connects with a CD Key that appears to be in use) the original server is contacted to double check that the user is still connected. If the original server doesn't respond, or responds with a negative, the new user is allowed to connect. Please note that this "double check" is handled entirely by the API code, so if you don't notify the API of a user disconnecting, the API will assume the user is still online.

Query and Reporting 2 SDK Integration

As mentioned in the "How it Works" section, you have the option of integrating the CDKey SDK with the Query and Reporting 2 SDK for additional security on the game server. This integration causes the CDKey SDK to use the networking code in the Query and Reporting 2 SDK for all incoming and outgoing data.

This provides two additional benefits to security:

- Any attempt to disable the CDKey validation code inside the server binary will likely result in the disabling of the Query and Reporting code - thus causing the server to not be listed on the master server list.
- When the CDKey network code is integrated with the Query and Reporting code, our backend can send special queries to the game server to verify that it is authenticating CDKeys correctly. If these checks fail, the server can be banned from the master server list automatically.

Both of these features help prevent people from running public, "cracked" servers that allow all clients to play on them without a valid CD Key. It is still possible for someone to run a "private" cracked server by blocking all network traffic to GameSpy's backend. However, that is really no different than if they were running a LAN server with no Internet access - CDKey validation would be disabled in that case anyway. Preventing cracked servers from being listed on the master server will make it nearly impossible for casual players to find any.

Enabling the Query & Reporting 2 integration is simple, and you should generally enable it unless you have a specific reason not to.

To enable the integration:

1. Define the pre-compiler directive "[QR2CDKEY_INTEGRATION](#)" when compiling the CD Key SDK. You can add this to the *gcdkeys.h* file, or as a compiler option.
2. Call [gcd_init_qr2](#) instead of [gcd_init](#) when initializing the CD

Key SDK. You will need to initialize the Query and Reporting 2 SDK prior to calling `gcd_init_qr2`.

Finally, if you are using the Query and Reporting NAT proxy support to share a socket between your game and the Query and Reporting 2 SDK, you will need to pass all CDKey network traffic to the `qr2_parse_query` function in addition to the normal QR2 traffic. You can identify CD Key network traffic by the first byte, which is always `0x3B` ("`;`").

CD Key Client SDK Functions

[gcd_compute_response](#)

Calculates a response to a challenge string.

gcd_compute_response

Calculates a response to a challenge string.

```
void gcd_compute_response(  
    char * cdkey,  
    char * challenge,  
    char response[73],  
    CDResponseMethod method );
```

Routine	Required Header	Distribution
gcd_compute_response	<gcdkeys.h>	SDKZIP

Parameters

cdkey

[in] The client's CD key.

challenge

[in] The challenge string. Should be no more than 32 characters.

response

[out] Receives the computed response string.

method

[in] Enum listing the response method - set to either `CDResponseMethod_NEWAUTH` or `CDResponseMethod_REAUTH`.

Remarks

When the client receives the challenge string it should calculate a response using the **gcd_compute_response** function in the Client API. Pass the client's CD key and the challenge string into the function and it will return the response string, a 72 character ASCII string. Send this response back to the server.

Section Reference: [Gamespy CDKey SDK](#)

CD Key Server SDK Functions

gcd_authenticate_user	Sends an authentication request.
gcd_disconnect_all	Calls gcd_disconnect_user for each user still online.
gcd_disconnect_user	Notify the validation server that a user has disconnected.
gcd_getkeyhash	Returns the key hash for the given user.
gcd_init	Initializes the Server API and creates the sockets and structures.
gcd_init_qr2	Initializes the Server API and integrates the networking of the CDKey SDK with the Query & Reporting 2 SDK.
gcd_process_reauth	Used to respond to a reauthentication request made by the validation server proving the client is still on.
gcd_shutdown	Release the socket and send disconnect messages to the validation server for any clients still on the server.
gcd_think	Processes any pending data from the validation server and calls the callback to indicate whether a client was authorized or not.



gcd_authenticate_user

Sends an authentication request.

```
void gcd_authenticate_user(  
    int gameid,  
    int localid,  
    unsigned int userip,  
    char * challenge,  
    char * response,  
    AuthCallbackFn authfn,  
    RefreshAuthCallbackFn refreshfn,  
    void * instance );
```

Routine	Required Header	Distribution
gcd_authenticate_user	<gcdkeys.h>	SDKZIP

Parameters

gameid

[in] The game ID issued for your game.

localid

[in] A unique int used to identify each client on the server. No two clients should have the same localid.

userip

[in] The client's IP address, preferably in network byte order.

challenge

[in] The challenge string that was sent to the client. Should be no more than 32 characters.

response

[in] The response that the client received.

authfn

[in] A callback that is called when the user is either authorized or rejected.

refreshfn

[in] A callback called when the server needs to re-authorize a client on the local host

instance

[in] Optional free-format user data for use by the callback.

Section Reference: [Gamespy CDKey SDK](#)

gcd_disconnect_all

Calls gcd_disconnect_user for each user still online.

```
void gcd_disconnect_all(  
    int gameid );
```

Routine	Required Header	Distribution
gcd_disconnect_all	<gcdkeys.h>	SDKZIP

Parameters

gameid

[in] The game ID issued for your game.

Section Reference: [Gamespy CDKey SDK](#)

gcd_disconnect_user

Notify the validation server that a user has disconnected.

```
void gcd_disconnect_user(  
    int gameid,  
    int localid );
```

Routine	Required Header	Distribution
gcd_disconnect_user	<gcdkeys.h>	SDKZIP

Parameters

gameid

[in] The game ID issued for your game.

localid

[in] The unique int used to identify the user.

Section Reference: [Gamespy CDKey SDK](#)

gcd_getkeyhash

Returns the key hash for the given user.

```
char * gcd_getkeyhash(  
    int gameid,  
    int localid );
```

Routine	Required Header	Distribution
gcd_getkeyhash	<gcdkeys.h>	SDKZIP

Return Value

Returns the key hash string, or an empty string if that user is not connected.

Parameters

gameid

[in] The game ID issued for your game.

localid

[in] The unique int used to identify the user.

Remarks

The hash returned will always be the same for a given user. This makes it useful for banning or tracking of users (used with the Tracking/Stats SDK).

Section Reference: [Gamespy CDKey SDK](#)

gcd_init

Initializes the Server API and creates the sockets and structures.

```
int gcd_init(  
    int gameid );
```

Routine	Required Header	Distribution
gcd_init	<gcdkeys.h>	SDKZIP

Return Value

Returns 0 if successful; non-zero if error.

Parameters

gameid

[in] The Game ID issued for your game.

Section Reference: [Gamespy CDKey SDK](#)

See Also: [gcd_init_qr2](#)

gcd_init_qr2

Initializes the Server API and integrates the networking of the CDKey SDK with the Query & Reporting 2 SDK.

```
int gcd_init_qr2(  
    qr2_t qrec,  
    int gameid );
```

Routine	Required Header	Distribution
gcd_init_qr2	<gcdkeys.h>	SDKZIP

Return Value

Returns 0 if successful; non-zero if error.

Parameters

qrec

[in] The initialized QR2 SDK object.

gameid

[in] The game ID issued for your game.

Remarks

You must initialize the Query & Reporting 2 SDK with `qr2_init` or `qr2_init_socket` prior to calling this. If you are using multiple instances of the QR2 SDK, you can pass the specific instance information in via the "qrec" argument. Otherwise you can simply pass in NULL.

Section Reference: [Gamespy CDKey SDK](#)

See Also: [gcd_init](#)

gcd_process_reauth

Used to respond to a reauthentication request made by the validation server proving the client is still on.

```
void gcd_process_reauth(  
    int gameid,  
    int localid,  
    int skey,  
    const char * response );
```

Routine	Required Header	Distribution
gcd_process_reauth	<gcdkeys.h>	SDKZIP

Parameters

gameid

[in] The game ID used to initialize the SDK with

localid

[in] An index of the client

skey

[in] The client's session key that came from the validation server

response

[in] The client's response to the challenge

Remarks

When the Reauthentication callback (passed to `gcd_authenticate user`) is called, the host/server must send the required information to verify that the client is still online, using the CD Key being checked. This should be called after the client has computed a response to the challenge coming from the callback.

Section Reference: [Gamespy CDKey SDK](#)

gcd_shutdown

Release the socket and send disconnect messages to the validation server for any clients still on the server.

void gcd_shutdown();

Routine	Required Header	Distribution
gcd_shutdown	<gcdkeys.h>	SDKZIP

Section Reference: [Gamespy CDKey SDK](#)

gcd_think

Processes any pending data from the validation server and calls the callback to indicate whether a client was authorized or not.

void gcd_think();

Routine	Required Header	Distribution
gcd_think	<gcdkeys.h>	SDKZIP

Remarks

This function should be called at least once every 10-100ms and is guaranteed not to block (although it may make a callback if an authorization response has come in). If your game uses the Query and Reporting SDK, you can place this call in the same area as the call to `qr_process_queries`.

Section Reference: [Gamespy CDKey SDK](#)

CD Key Server SDK Callbacks

[AuthCallBackFn](#)

Called when the user is either authorized or rejected.

[RefreshAuthCallBackFn](#)

Used to reauthenticate a client for the purpose of proving a client is still online.

AuthCallbackFn

Called when the user is either authorized or rejected.

```
typedef void (*AuthCallbackFn)(  
    int gameid,  
    int localid,  
    int authenticated,  
    char * errmsg,  
    void * instance );
```

Routine	Required Header	Distribution
AuthCallbackFn	<gcdkeys.h>	SDKZIP

Parameters

gameid

[in] The game ID for which authentication is requested.

localid

[in] The id that was passed into `gcd_authenticate_user`.

authenticated

[in] Indicates whether the user was authenticated: 1 if authenticated; 0 if not.

errmsg

[in] Error message if user was not authenticated.

instance

[in] The same instance as was passed into the `gcd_authenticate_user`.

Remarks

This function will be called within two seconds of `gcd_authenticate_user`, even if the validation server hasn't responded yet.

If the user was not authenticated, the `errmsg` parameter contains a descriptive string of the reason (either CD Key not valid, or CD Key in use).

Section Reference: [Gamespy CDKey SDK](#)

RefreshAuthCallbackFn

Used to reauthenticate a client for the purpose of proving a client is still online.

```
typedef void (*RefreshAuthCallbackFn)(  
    int gameid,  
    int localid,  
    int hint,  
    char * challenge,  
    void * instance );
```

Routine	Required Header	Distribution
RefreshAuthCallbackFn	<gcdkeys.h>	SDKZIP

Parameters

gameid

[in] the game ID used to initialize the SDK with

localid

[in] the index of the player

hint

[in] a session id for a client used for reauthentication - this is the skey passed into gcd_process_reauth

challenge

[in] a challenge string used for reauthentication

instance

[in] user data passed in gcd_authenticate_user

Remarks

The reauthentication callback will be called any time the validation server wishes to determine if a client is still online. When called, the client index, challenge, and session key will be available. These values must be used to reauthenticate the user. Remember that this process is similar to the primary authentication process, where the only difference is that the validation server provides the challenge and session key (note: the "hint" parameter in this callback is the session key that should be passed as the "skey" value into `gcd_process_reauth`).

Section Reference: [Gamespy CDKey SDK](#)

CD Key SDK Enumerations

[CDResponseMethod](#)

Values are passed to the `gcd_compute_response` function done client side.

CDResponseMethod

Values are passed to the `gcd_compute_response` function done client side.

```
typedef enum  
{  
    CDResponseMethod_NEWAUTH,  
    CDResponseMethod_REAUTH  
} CDResponseMethod;
```

Constants

CDResponseMethod_NEWAUTH

Used for primary authentications.

CDResponseMethod_REAUTH

Used for re-authentications.

Section Reference: [Gamespy CDKey SDK](#)

Chat SDK

Overview

The GameSpy Chat SDK is a portable ANSI-C API used to write chat clients. The current implementation works with IRC servers, however it could be re-implemented to work on another chat network without having to change any user code. The Chat SDK provides an easy way to allow your game's players to chat online. There are no libraries or DLLs to deal with; just add the source files to your project and you're ready to go.

The Chat SDK only deals with data. You will be responsible for creating all the GUI elements that are required for chatting within your game.

Chat Nicknames have a few restrictions based on IRC standards and server requirements. The character limit for chat nicks is 20 characters. The following are the character restrictions:

- The first character cannot be any of the following: +, @, #, :
- Numeric characters are only allowed after the first character.
- All characters in the ASCII character range 34-126 are valid except for character 92.

File Manifest

The following files should be included with this package. If any of the files are missing, please [contact devsupport@gamespy.com](mailto:devsupport@gamespy.com).

File	Description
<i>chat.h</i> prototyped here)	GameSpy Chat header (all user functions are
<i>chatMain.c</i>	Entry point for all user Chat functions
<i>chatMain.h</i>	Common header for internal code
<i>chatSocket.c</i> a chat server	Implementation of a network-level connection to
<i>chatSocket.h</i>	Header for chat socket functions
<i>chatHandlers.c</i>	Code for handling IRC messages
<i>chatHandlers.h</i>	Header for callback handling functions
<i>chatCallbacks.c</i>	Code for queueing and calling callbacks
<i>chatCallbacks.h</i>	Header for callback handling function
<i>chatChannel.c</i> users in the channels	Code for dealing with chat channels and the
<i>chatChannel.h</i> channel and user data	Header for accessing and manipulating the
<i>nonport.c</i>	Platform-specific code
<i>nonport.h</i>	Platform-specific header
<i>hashtable.c</i>	Hashtable implementation
<i>hashtable.h</i>	Hashtable headers
<i>darray.c</i>	Dynamic-Array implementation
<i>darray.h</i>	Dynamic-Array headers

Implementation

Connecting

The first thing to do with Chat is to connect to a server. This is done with either `chatConnect`, `chatConnectSpecial`, or `chatConnectSecure`. Most application will use `chatConnect`. `chatConnectSpecial` is used to fill in the user field after the local machine's IP address is known, and `chatConnectSecure` is used to encrypt the connection.

```
CHAT chatConnect(const char * serverAddress, int port,
const char * nick, const char * user, const char *
name, chatGlobalCallbacks * callbacks,
chatNickErrorCallback nickErrorCallback,
chatConnectCallback connectCallback, void * param,
CHATBool blocking)
```

```
CHAT chatConnectSpecial(const char * serverAddress,
int port, const char * nick, const char * name,
chatGlobalCallbacks * callbacks, chatNickErrorCallback
nickErrorCallback, chatFillInUserCallback
fillInUserCallback, chatConnectCallback
connectCallback, void * param, CHATBool blocking)
```

```
CHAT chatConnectSecure(const char * serverAddress, int
port, const char * nick, const char * name, const char
* gamename, const char * secretKey,
chatGlobalCallbacks * callbacks, chatNickErrorCallback
nickErrorCallback, chatFillInUserCallback
fillInUserCallback, chatConnectCallback
connectCallback, void * param, CHATBool blocking)
```

`serverAddress`

the IP address and port of the chat server to which to connect

`port`

the port of the chat server to which to connect

nick

the connecting user's nickname

user

the user's username. This is only used with `chatConnect`.

name

the user's real name or any other optional info

gamename, secret key

used with `chatConnectSecure`, which is used to encrypt all traffic with the chat server.

The gamename and secretKey are application-specific - if you are unsure what your gamename and secretKey are, contact devsupport@gamespy.com.

callbacks

a pointer to a structure which contains a list of global callbacks to be associated with this connection. The structure also contains a "param" member which is of type pointer to `void (void *)`. This param is passed in as the last argument to all global callbacks.

`chatConnect` returns a CHAT object. This represents the connection to the chat server. If the return value is NULL, then there was an error establishing the connection.

Connecting should look something like this:

```
int CMyGame::OnConnect(...)
{
    m_chat = chatConnect("irc.mygame.com", 6667,
        if(m_chat == NULL)
            Error();
}
```

Disconnecting

When the chat connection is ready to be disconnected, just call the `chatDisconnect` function:

```
void chatDisconnect(CHAT chat)
```

This will terminate the connection to the chat server. The chat object cannot be used again. To establish a new connection, `chatConnect` must be called again. `chatDisconnect` should always be called to cleanup a connection - the only exception is when `chatConnect` returns `NULL`.

Processing

A chat connection must be periodically processed. This is done by calling `chatThink`:

```
void chatThink(CHAT chat)
```

When a connection is processed, it sends any queued outgoing data, reads incoming data, and calls any callbacks generated by the incoming data. This function can be called in an applications main or idle loop. It should be called at least once a second, but it is not necessary to call it more than several times a second (although calling it more often will do no harm).

Entering A Channel

To join a channel, call `chatEnterChannel`. This function will enter an existing channel if it exists or create a new channel and enter it if it does not exist.

```
void chatEnterChannel(CHAT chat, const char * channel,
```

`chat`

the same `CHAT` object returned by the call to `chatConnect`

`channel`

the channel that we are trying to enter

`password`

the password required to enter the channel. If no password is

required, this can either be `NULL` or an empty string.

`blocking`

determines if this function should block until the enter attempt has been completed or if it should be returned immediately. In either case, "callback" will be called when the attempt is completed.

Leaving A Channel

To leave a channel, just call `chatLeaveChannel`:

```
void chatLeaveChannel(CHAT chat, const char * channel)
```

This will take you out of the given channel.

UNICODE Support

The GameSpy SDKs support an optional UNICODE interface for widestring applications. To use this interface, first define the symbol "[GSI_UNICODE](#)". Then, use widestrings wherever ANSI strings were previously called for. When in doubt, please refer to the header files for specific function declarations.

Although the GameSpy SDK interfaces support UNICODE parameters, some items may be stripped of their extra UNICODE information. These items include: nickname, email address, and URL strings. You may pass in widestring values, but they will first be converted to their ANSI counterparts before transmission.

Chat SDK Functions

chatAddChannelBan	Ban a nickname from the specified channel. Local client must have moderator privileges.
chatAuthenticateCDKey	Allows pre-chat cd key authentication via the chat server.
chatBanUser	Ban a user from the chat room. The user may not rejoin.
chatChangeNick	Change the chat nickname associated with the local client. This does not affect the account name.
chatConnect	The chatConnect function initializes the Chat SDK and initiates a connection to the chat server.
chatConnectLogin	Initializes the Chat SDK and initiates a connection to the chat server. The chatConnectLogin function provides the ability to login to chat using a registered unique nickname.
chatConnectPreAuth	Initializes the Chat SDK and initiates a connection to the chat server. The chatConnectPreAuth function provides the ability to specify authtoken and partnerchallenge. (Not for common use).

chatConnectSecure	Initializes the Chat SDK and initiates a connection to the chat server. The chatConnectSecure function encrypts the connection.
chatConnectSpecial	Initializes the Chat SDK and initiates a connection to the chat server. The chatConnectSpecial function provides ability to fill in the user field after the local machine's IP address is known.
chatDisconnect	Disconnect from the chat server. Performs necessary cleanup of the Chat SDK
chatEnterChannel	Joins a chat channel.
chatEnumChannelBans	Retrieves a list of clients banned from a channel.
chatEnumChannels	Enumerates the chat channels on the server.
chatEnumJoinedChannels	Enumerates the chat channels on the server which the local client has joined.
chatEnumUsers	Retrieves the list of users in the specified channel.
chatFixNick	Repairs an illegal chat nickname.
chatGetBasicUserInfo	

	Retrieves basic information on the specified user.
chatGetBasicUserInfoNoWait	Retrieves basic information on the specified user. Information is returned through function parameters.
chatGetChannelBasicUserInfo	Retrieves basic user info for every member of the specified channel.
chatGetChannelKeys	Retrieves a list of key/value pairs for a channel or user.
chatGetChannelMode	Retrieves the "mode" of a channel.
chatGetChannelNumUsers	Returns the number of users in the already joined channel. This is a cached value, and not a server query.
chatGetChannelPassword	Queries the server for the specified channel's password.
chatGetChannelTopic	Queries the server for the specified channel's topic. Also known as the room description.
chatGetGlobalKeys	Retrieves a list of global keys for a single user, or all users.
chatGetNick	Gets the chat nickname of the local client. This may not be the same as the profile

	nickname.
chatGetProfileID	Gets the profile id of the local client.
chatGetUserID	Gets the user id of the local client.
chatGetUserInfo	Gets information on the specified user.
chatGetUserMode	Get the mode of a user in a specified channel.
chatGetUserModeNoWait	Get the mode of a user in a specified channel, returning it through a function parameter.
chatInChannel	Determine whether the local client is a member of the specified channel.
chatInviteUser	Invite a user to join a channel.
chatKickUser	Forcefully remove a user from a specified channel.
chatLeaveChannel	Leave a chat channel.
chatRegisterUniqueNick	Registers a unique nick to the local client and cdkey.
chatRemoveChannelBan	Removes a banned player from a

	channel's ban list. This will once again allow the user to join the channel.
chatRetryWithNick	Use in response to a nickErrorCallback. This function allows the local client to retry the connection attempt with a different chat nickname.
chatSendChannelMessage	Send a message to all members of the specified channel.
chatSendRaw	Send a raw command to the chat server. This does not automatically send to a player.
chatSendUserMessage	Send a private message to a user.
chatSetChannelGroup	Assign a user-defined grouping to a channel. The group is a string identifier which is linked to the channel.
chatSetChannelKeys	Set key/values on a channel or the local user.
chatSetChannelLimit	Set the maximum number of users allowed in a channel.
chatSetChannelMode	Set a channel's mode.
chatSetChannelPassword	Sets or clears a password on the specified channel.

chatSetChannelTopic	Set the topic (description) of a chat channel.
chatSetGlobalKeys	Set key/values on the local client.
chatSetQuietMode	Sets the chat sdk to quiet mode or disables quiet mode.
chatSetUserMode	Set the IRC mode of the specified user. This mode is applied in the specified channel.
chatThink	Allow the Chat SDK to continue processing.
chatTranslateNick	Removes the namespace extension from a nickname. Use this when working with unique nicknames in a public chat room.

chatAddChannelBan

Ban a nickname from the specified channel. Local client must have moderator privileges.

```
void chatAddChannelBan(  
    CHAT chat,  
    const gsi_char * channel,  
    const gsi_char * ban );
```

Routine	Required Header	Distribution
chatAddChannelBan	<chat.h>	SDKZIP

Parameters

chat

[in] Chat SDK object, previously initialized using one of the chatConnect methods.

channel

[in] Name of chat channel from which user is being banned.

ban

[in] Chat nickname of user being banned.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
chatAddChannelBan	chatAddChannelBanA	chatAddChannelBanW

chatAddChannelBanW and **chatAddChannelBanA** are UNICODE and ANSI mapped versions of **chatAddChannelBan**. The arguments of **chatAddChannelBanA** are ANSI strings; those of **chatAddChannelBanW** are wide-character strings.

Section Reference: [Gamespy Chat SDK](#)

chatAuthenticateCDKey

Allows pre-chat cd key authentication via the chat server.

```
void chatAuthenticateCDKey(  
    CHAT chat,  
    const gsi_char * cdkey,  
    chatAuthenticateCDKeyCallback callback,  
    void * param,  
    CHATBool blocking );
```

Routine	Required Header	Distribution
chatAuthenticateCDKey	<chat.h>	SDKZIP

Parameters

chat

[in] Chat SDK object, previously initialized using one of the chatConnect methods.

cdkey

[in] CD key to validate; should be a valid CD key for the set game title.

callback

[in] Optional user-supplied function to be called when the operation has completed.

param

[in] Optional pointer to user data; will be passed unmodified to the callback function.

blocking

[in] If CHATTrue, return only after the operation has completed; otherwise, return immediately.

Remarks

The **chatAuthenticateCDKey** function may be used to authenticate a user's cdkey before they enter the chat room. This should not be a substitute for a cdkey during gameplay. Arcade does not support this call, so users in Arcade will be able to enter chat without this validation. This method most usefull for developers who opt-out of the Arcade compatability requirements or have a separate chat area for in-game clients.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
chatAuthenticateCDKey	chatAuthenticateCDKeyA	chatAuthenticateCDKeyW

chatAuthenticateCDKeyW and **chatAuthenticateCDKeyA** are UNICODE and ANSI mapped versions of **chatAuthenticateCDKey**. The arguments of **chatAuthenticateCDKeyA** are ANSI strings; those of **chatAuthenticateCDKeyW** are wide-character strings.

Section Reference: [Gamespy Chat SDK](#)

See Also: [ChatConnect](#)

chatBanUser

Ban a user from the chat room. The user may not rejoin.

```
void chatBanUser(  
    CHAT chat,  
    const gsi_char * channel,  
    const gsi_char * user );
```

Routine	Required Header	Distribution
chatBanUser	<chat.h>	SDKZIP

Parameters

chat

[in] Chat SDK object, previously initialized using one of the chatConnect methods.

channel

[in] Name of chat channel from which user is being banned.

user

[in] Chat nickname of user being banned.

Remarks

The caller of this function must have operator privileges for the channel in which the ban is to be performed.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
chatBanUser	chatBanUserA	chatBanUserW

chatBanUserW and **chatBanUserA** are UNICODE and ANSI mapped versions of **chatBanUser**. The arguments of **chatBanUserA** are ANSI strings; those of **chatBanUserW** are wide-character strings.

Section Reference: [Gamespy Chat SDK](#)

See Also: [ChatConnect](#)

chatChangeNick

Change the chat nickname associated with the local client. This does not affect the account name.

```
void chatChangeNick(  
    CHAT chat,  
    const gsi_char * newNick,  
    chatChangeNickCallback callback,  
    void * param,  
    CHATBool blocking );
```

Routine	Required Header	Distribution
chatChangeNick	<chat.h>	SDKZIP

Parameters

chat

[in] Chat SDK object, previously initialized using one of the chatConnect methods.

newNick

[in] Nickname to assign to the local user.

callback

[in] Optional user-supplied function to be called when the operation has completed.

param

[in] Optional pointer to user data; will be passed unmodified to the callback function.

blocking

[in] If CHATTrue, return only after the operation has completed; otherwise, return immediately.

Remarks

The **chatChangeNick** function may be used to change a user's nickname as it appears in chat. This has no affect on GameSpy profile names such as those used for presence detection and buddy lists. Only one instance of a nickname may be in use at a time.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
chatChangeNick	chatChangeNickA	chatChangeNickW

chatChangeNickW and **chatChangeNickA** are UNICODE and ANSI mapped versions of **chatChangeNick**. The arguments of **chatChangeNickA** are ANSI strings; those of **chatChangeNickW** are wide-character strings.

Section Reference: [Gamespy Chat SDK](#)

chatConnect

The chatConnect function initializes the Chat SDK and initiates a connection to the chat server.

```
CHAT chatConnect(  
    const gsi_char * serverAddress,  
    int port,  
    const gsi_char * nick,  
    const gsi_char * user,  
    const gsi_char * name,  
    chatGlobalCallbacks * callbacks,  
    chatNickErrorCallback nickErrorCallback,  
    chatConnectCallback connectCallback,  
    void * param,  
    CHATBool blocking );
```

Routine	Required Header	Distribution
chatConnect	<chat.h>	SDKZIP

Return Value

This function returns the initialized Chat SDK interface. A return value of NULL indicates an error.

Parameters

serverAddress

[in] Address of the chat server being connect to; usually "peerchat.gamespy.com".

port

[in] Port of the chat server; usually 6667.

nick

[in] Nickname in use while chatting. Not associated with a user account in any way.

user

[in] User's username

name

[in] User's real name, or any other optional info.

callbacks

[in] Structure for specifying global handlers.

nickErrorCallback

[in] Optional user-supplied function to be called if nickname is invalid or in use.

connectCallback

[in] Optional user-supplied function to be called when the operation has completed.

param

[in] Optional pointer to user data; will be passed unmodified to the callback function.

blocking

[in] If CHATTrue, return only after the operation has completed; otherwise, return immediately.

Remarks

The server address and port for the connect functions can be left empty. In other words, serverAddress can be NULL, and the port can be specified to be 0. The SDK will automatically take care of using the default address and port.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
chatConnect	chatConnectA	chatConnectW

chatConnectW and **chatConnectA** are UNICODE and ANSI mapped versions of **chatConnect**. The arguments of **chatConnectA** are ANSI strings; those of **chatConnectW** are wide-character strings.

Example

```
int CMyGame::OnConnect(...)
{
    m_chat = chatConnect("irc.mygame.com", 6667, "nick", "user", "er
    if (m_chat == NULL)
        Error();
}
```

Section Reference: [Gamespy Chat SDK](#)

See Also: [chatConnectLogin](#), [chatConnectPreAuth](#), [chatConnectSecure](#), [chatConnectSpecial](#)

chatConnectLogin

Initializes the Chat SDK and initiates a connection to the chat server. The chatConnectLogin function provides the ability to login to chat using a registered unique nickname.

```
CHAT chatConnectLogin(  
    const gsi_char * serverAddress,  
    int port,  
    int namespaceID,  
    const gsi_char * email,  
    const gsi_char * profilenick,  
    const gsi_char * uniquenick,  
    const gsi_char * password,  
    const gsi_char * name,  
    const gsi_char * gamename,  
    const gsi_char * secretKey,  
    chatGlobalCallbacks * callbacks,  
    chatNickErrorCallback nickErrorCallback,  
    chatFillInUserCallback fillInUserCallback,  
    chatConnectCallback connectCallback,  
    void * param,  
    CHATBool blocking );
```

Routine	Required Header	Distribution
chatConnectLogin	<chat.h>	SDKZIP

Return Value

This function returns the initialized Chat SDK interface. A return value of NULL indicates an error.

Parameters

serverAddress

[in] Address of the chat server being connect to; usually "peerchat.gamespy.com".

port

[in] Port of the chat server; usually 6667.

namespaceID

[in] ID of the unique name namespace in which the users nickname is registered.

email

[in] E-mail address of the local client's GameSpy profile.

profilenick

[in] Nickname used when creating profile. May be different from the registered unique nick.

uniquenick

[in] Unique nickname registered to the profile with which user is logging in.

password

[in] Password of the GameSpy profile.

name

[in] User's real name, or any other optional info.

gamename

[in] Assigned gamename from which the local client is logging in.

secretKey

[in] Assigned secret key for the specified gamename.

callbacks

[in] Structure for specifying global handlers.

nickErrorCallback

[in] Optional user-supplied function to be called if nickname is invalid or in use.

fillInUserCallback

[in] Optional user-supplied function to be called when the SDK

requires the user name.

connectCallback

[in] Optional user-supplied function to be called when the operation has completed.

param

[in] Optional pointer to user data; will be passed unmodified to the callback function.

blocking

[in] If CHATTrue, return only after the operation has completed; otherwise, return immediately.

Remarks

The server address and port for the connect functions can be left empty. In other words, serverAddress can be NULL, and the port can be specified to be 0. The SDK will automatically take care of using the default address and port.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
chatConnectLogin	chatConnectLoginA	chatConnectLoginW

chatConnectLoginW and **chatConnectLoginA** are UNICODE and ANSI mapped versions of **chatConnectLogin**. The arguments of **chatConnectLoginA** are ANSI strings; those of **chatConnectLoginW** are wide-character strings.

Section Reference: [Gamespy Chat SDK](#)

See Also: [chatConnect](#), [chatConnectPreAuth](#), [chatConnectSecure](#), [chatConnectSpecial](#)

chatConnectPreAuth

Initializes the Chat SDK and initiates a connection to the chat server. The chatConnectPreAuth function provides the ability to specify authtoken and partnerchallenge. (Not for common use).

```
CHAT chatConnectPreAuth(  
    const gsi_char * serverAddress,  
    int port,  
    const gsi_char * authtoken,  
    const gsi_char * partnerchallenge,  
    const gsi_char * name,  
    const gsi_char * gamename,  
    const gsi_char * secretKey,  
    chatGlobalCallbacks * callbacks,  
    chatNickErrorCallback nickErrorCallback,  
    chatFillInUserCallback fillInUserCallback,  
    chatConnectCallback connectCallback,  
    void * param,  
    CHATBool blocking );
```

Routine	Required Header	Distribution
chatConnectPreAuth	<chat.h>	SDKZIP

Return Value

This function returns the initialized Chat SDK interface. A return value of NULL indicates an error.

Parameters

serverAddress

[in] Address of the chat server to connect to; usually "peerchat.gamespy.com".

port

[in] Port of the chat server; usually 6667.

authtoken

[in] Authentication token for this login.

partnerchallenge

[in] Partner challenge for this login.

name

[in] The user's real name, or any other optional info.

gamename

[in] GameName of the title this client is connecting from.

secretKey

[in] Assigned secret key for the specified gamename.

callbacks

[in] Structure for specifying global handlers.

nickErrorCallback

[in] Optional user-supplied function to be called if nickname is invalid or in use.

fillInUserCallback

[in] Optional user-supplied function to be called when the SDK requires the user name.

connectCallback

[in] Optional user-supplied function to be called when the operation has completed.

param

[in] Optional pointer to user data; will be passed unmodified to the callback function.

blocking

[in] If CHATTrue, return only after the operation has completed;

otherwise, return immediately.

Remarks

The server address and port for the connect functions can be left empty. In other words, serverAddress can be NULL, and the port can be specified to be 0. The SDK will automatically take care of using the default address and port.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
chatConnectPreAuth	chatConnectPreAuthA	chatConnectPreAuthW

chatConnectPreAuthW and **chatConnectPreAuthA** are UNICODE and ANSI mapped versions of **chatConnectPreAuth**. The arguments of **chatConnectPreAuthA** are ANSI strings; those of **chatConnectPreAuthW** are wide-character strings.

Section Reference: [Gamespy Chat SDK](#)

See Also: [chatConnect](#), [chatConnectLogin](#), [chatConnectSecure](#), [chatConnectSpecial](#)

chatConnectSecure

Initializes the Chat SDK and initiates a connection to the chat server. The chatConnectSecure function encrypts the connection.

```
CHAT chatConnectSecure(  
    const gsi_char * serverAddress,  
    int port,  
    const gsi_char * nick,  
    const gsi_char * name,  
    const gsi_char * gamename,  
    const gsi_char * secretKey,  
    chatGlobalCallbacks * callbacks,  
    chatNickErrorCallback nickErrorCallback,  
    chatFillInUserCallback fillInUserCallback,  
    chatConnectCallback connectCallback,  
    void * param,  
    CHATBool blocking );
```

Routine	Required Header	Distribution
chatConnectSecure	<chat.h>	SDKZIP

Return Value

Returns the initialized Chat SDK interface. A return value of NULL indicates an error.

Parameters

serverAddress

[in] Address of the chat server to connect to; usually "peerchat.gamespy.com".

port

[in] Port of the chat server; usually 6667.

nick

[in] Nickname in use while chatting. Not associated with a user account in any way.

name

[in] User's real name, or any other optional info.

gamename

[in] GameName of the title this client is connecting from.

secretKey

[in] Assigned secret key for the specified gamename.

callbacks

[in] Structure for specifying global handlers.

nickErrorCallback

[in] Optional user-supplied function to be called if nickname is invalid or in use.

fillInUserCallback

[in] Optional user-supplied function to be called when the SDK requires the user name.

connectCallback

[in] Optional user-supplied function to be called when the operation has completed.

param

[in] Optional pointer to user data; will be passed unmodified to the callback function.

blocking

[in] If CHATTrue, return only after the operation has completed; otherwise, return immediately.

Remarks

The server address and port for the connect functions can be left empty. In other words, serverAddress can be NULL, and the port can be specified to be 0. The SDK will automatically take care of using the default address and port.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
chatConnectSecure	chatConnectSecureA	chatConnectSecureW

chatConnectSecureW and **chatConnectSecureA** are UNICODE and ANSI mapped versions of **chatConnectSecure**. The arguments of **chatConnectSecureA** are ANSI strings; those of **chatConnectSecureW** are wide-character strings.

Section Reference: [Gamespy Chat SDK](#)

See Also: [chatConnect](#), [chatConnectLogin](#), [chatConnectPreAuth](#), [chatConnectSpecial](#)

chatConnectSpecial

Initializes the Chat SDK and initiates a connection to the chat server. The chatConnectSpecial function provides ability to fill in the user field after the local machine's IP address is known.

```
CHAT chatConnectSpecial(  
    const gsi_char * serverAddress,  
    int port,  
    const gsi_char * nick,  
    const gsi_char * name,  
    chatGlobalCallbacks * callbacks,  
    chatNickErrorCallback nickErrorCallback,  
    chatFillInUserCallback fillInUserCallback,  
    chatConnectCallback connectCallback,  
    void * param,  
    CHATBool blocking );
```

Routine	Required Header	Distribution
chatConnectSpecial	<chat.h>	SDKZIP

Return Value

This function returns the initialized Chat SDK interface. A return value of NULL indicates an error.

Parameters

serverAddress

[in] Address of the chat server to connect to; usually "peerchat.gamespy.com".

port

[in] Port of the chat server; usually 6667.

nick

[in] Nickname in use while chatting. Not associated with a user account in any way.

name

[in] User's real name, or any other optional info.

callbacks

[in] Structure for specifying global handlers.

nickErrorCallback

[in] Callback that is triggered if nick is invalid or in use.

fillInUserCallback

[in] Optional user-supplied function to be called when the SDK requires the user name.

connectCallback

[in] Optional user-supplied function to be called when the connection attempt has completed.

param

[in] Optional pointer to user data; will be passed unmodified to the callback function.

blocking

[in] If CHATTrue, return only after the operation has completed; otherwise, return immediately.

Remarks

The server address and port for the connect functions can be left empty. In other words, serverAddress can be NULL, and the port can be specified to be 0. The SDK will automatically take care of using the default address and port.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
chatConnectSpecial	chatConnectSpecialA	chatConnectSpecialW

chatConnectSpecialW and **chatConnectSpecialA** are UNICODE and ANSI mapped versions of **chatConnectSpecial**. The arguments of **chatConnectSpecialA** are ANSI strings; those of **chatConnectSpecialW** are wide-character strings.

Section Reference: [Gamespy Chat SDK](#)

See Also: [chatConnect](#), [chatConnectLogin](#), [chatConnectPreAuth](#), [chatConnectSecure](#)

chatDisconnect

Disconnect from the chat server. Performs necessary cleanup of the Chat SDK.

```
void chatDisconnect(  
    CHAT chat );
```

Routine	Required Header	Distribution
chatDisconnect	<chat.h>	SDKZIP

Parameters

chat

[in] Chat SDK object, previously initialized using one of the chatConnect methods.

Remarks

The **chatDisconnect** function disconnects the SDK from the chat server and performs necessary cleanup on the CHAT object. The CHAT object is invalid after this call has completed. To continue using the chat SDK you must reinitialize using one of the chat connect methods.

Section Reference: [Gamespy Chat SDK](#)

See Also: [ChatConnect](#)

chatEnterChannel

Joins a chat channel.

```
void chatEnterChannel(  
    CHAT chat,  
    const gsi_char * channel,  
    const gsi_char * password,  
    chatChannelCallbacks * callbacks,  
    chatEnterChannelCallback callback,  
    void * param,  
    CHATBool blocking );
```

Routine	Required Header	Distribution
chatEnterChannel	<chat.h>	SDKZIP

Parameters

chat

[in] Chat SDK object, previously initialized using one of the chatConnect methods.

channel

[in] Name of the chat channel being joined.

password

[in] Password of the channel. Ignored if no password has been set.

callbacks

[in] Structure for specifying global handlers; for channel-specific traffic such as user messages.

callback

[in] Optional user-supplied function to be called when the operation has completed.

param

[in] Optional pointer to user data; will be passed unmodified to the callback function.

blocking

[in] If CHATTrue, return only after the operation has completed; otherwise, return immediately.

Remarks

The **chatEnterChannel** function is used to add the local client to a chat channel. If the channel is password protected the valid password must be supplied. If it is not, the callback will be triggered with an invalid password result.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
chatEnterChannel	chatEnterChannelA	chatEnterChannelW

chatEnterChannelW and **chatEnterChannelA** are UNICODE and ANSI mapped versions of **chatEnterChannel**. The arguments of **chatEnterChannelA** are ANSI strings; those of **chatEnterChannelW** are wide-character strings.

Section Reference: [Gamespy Chat SDK](#)

See Also: [ChatConnect](#), [ChatDisconnect](#)

chatEnumChannelBans

Retrieves a list of clients banned from a channel.

```
void chatEnumChannelBans(  
    CHAT chat,  
    const gsi_char * channel,  
    chatEnumChannelBansCallback callback,  
    void * param,  
    CHATBool blocking );
```

Routine	Required Header	Distribution
chatEnumChannelBans	<chat.h>	SDKZIP

Parameters

chat

[in] Chat SDK object, previously initialized using one of the chatConnect methods.

channel

[in] Name of the chat channel whose ban list is being retrieved.

callback

[in] Optional user-supplied function to be called when the operation has completed; will be passed the list of banned clients.

param

[in] Optional pointer to user data; will be passed unmodified to the callback function.

blocking

[in] If CHATTrue, return only after the operation has completed; otherwise, return immediately.

Remarks

The local client must have operator privileges to execute this command.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Define
chatEnumChannelBans	chatEnumChannelBansA	chatEnumChannelBans

chatEnumChannelBansW and **chatEnumChannelBansA** are UNICODE and ANSI mapped versions of **chatEnumChannelBans**. The arguments of **chatEnumChannelBansA** are ANSI strings; those of **chatEnumChannelBansW** are wide-character strings.

Section Reference: [Gamespy Chat SDK](#)

chatEnumChannels

Enumerates the chat channels on the server.

```
void chatEnumChannels(  
    CHAT chat,  
    const gsi_char * filter,  
    chatEnumChannelsCallbackEach callbackEach,  
    chatEnumChannelsCallbackAll callbackAll,  
    void * param,  
    CHATBool blocking );
```

Routine	Required Header	Distribution
chatEnumChannels	<chat.h>	SDKZIP

Parameters

chat

[in] Chat SDK object, previously initialized using one of the chatConnect methods.

filter

[in] String comparison used to filter the channel results. Example "#gsp!mygame!". Use the "*" for the wildcard.

callbackEach

[in] Optional user-supplied function to be called once for each channel in the list.

callbackAll

[in] Optional user-supplied function to be called once for the full channel list.

param

[in] Optional pointer to user data; will be passed unmodified to the callback function.

blocking

[in] If CHATTrue, return only after the operation has completed; otherwise, return immediately.

Remarks

The **chatEnumChannels** function enumerates the chat channels which match the current search criteria. Typical information returned on each channel includes the topic and number of users. The filter can contain wildcards used to get all channels when passing in a partial name and wildcard.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
chatEnumChannels	chatEnumChannelsA	chatEnumChannelsW

chatEnumChannelsW and **chatEnumChannelsA** are UNICODE and ANSI mapped versions of **chatEnumChannels**. The arguments of **chatEnumChannelsA** are ANSI strings; those of **chatEnumChannelsW** are wide-character strings.

Section Reference: [Gamespy Chat SDK](#)

See Also: [ChatConnect](#), [ChatDisconnect](#)

chatEnumJoinedChannels

Enumerates the chat channels on the server which the local client has joined. .

```
void chatEnumJoinedChannels(  
    CHAT chat,  
    chatEnumJoinedChannelsCallback callback,  
    void * param );
```

Routine	Required Header	Distribution
chatEnumJoinedChannels	<chat.h>	SDKZIP

Parameters

chat

[in] Chat SDK object, previously initialized using one of the chatConnect methods.

callback

[in] Optional user-supplied function to be called once for each channel in the list.

param

[in] Optional pointer to user data; will be passed unmodified to the callback function.

Remarks

For each channel, a channel index value is returned that may be used to retrieve further information about the channel.

Section Reference: [Gamespy Chat SDK](#)

See Also: [ChatConnect](#), [ChatDisconnect](#), [ChatEnumChannels](#)

chatEnumUsers

Retrieves the list of users in the specified channel.

```
void chatEnumUsers(  
    CHAT chat,  
    const gsi_char * channel,  
    chatEnumUsersCallback callback,  
    void * param,  
    CHATBool blocking );
```

Routine	Required Header	Distribution
chatEnumUsers	<chat.h>	SDKZIP

Parameters

chat

[in] Chat SDK object, previously initialized using one of the chatConnect methods.

channel

[in] Name of the chat channel whose user list is being retrieved.

callback

[in] Optional user-supplied function to be called when the operation has completed; will be passed the user list.

param

[in] Optional pointer to user data; will be passed unmodified to the callback function.

blocking

[in] If CHATTrue, return only after the operation has completed; otherwise, return immediately.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
chatEnumUsers	chatEnumUsersA	chatEnumUsersW

chatEnumUsersW and **chatEnumUsersA** are UNICODE and ANSI mapped versions of **chatEnumUsers**. The arguments of **chatEnumUsersA** are ANSI strings; those of **chatEnumUsersW** are wide-character strings.

Section Reference: [Gamespy Chat SDK](#)

chatFixNick

Repairs an illegal chat nickname.

```
void chatFixNick(  
    gsi_char * newNick,  
    const gsi_char * oldNick );
```

Routine	Required Header	Distribution
chatFixNick	<chat.h>	SDKZIP

Parameters

newNick

[out] Receives corrected nickname; may be identical to original nickname if no issues are detected.

oldNick

[in] Nickname to be corrected or verified.

Remarks

The **chatFixNick** function replaces illegal characters in the nickname with the underscore ("_") character. This function will also replace leading numbers and illegal whitespace combinations.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
chatFixNick	chatFixNickA	chatFixNickW

chatFixNickW and **chatFixNickA** are UNICODE and ANSI mapped versions of **chatFixNick**. The arguments of **chatFixNickA** are ANSI strings; those of **chatFixNickW** are wide-character strings.

Section Reference: [Gamespy Chat SDK](#)

See Also: [ChatConnect](#), [ChatDisconnect](#)

chatGetBasicUserInfo

Retrieves basic information on the specified user.

```
void chatGetBasicUserInfo(  
    CHAT chat,  
    const gsi_char * user,  
    chatGetBasicUserInfoCallback callback,  
    void * param,  
    CHATBool blocking );
```

Routine	Required Header	Distribution
chatGetBasicUserInfo	<chat.h>	SDKZIP

Parameters

chat

[in] Chat SDK object, previously initialized using one of the chatConnect methods.

user

[in] User's assigned GameName.

callback

[in] Optional user-supplied function to be called when the operation has completed; will be passed the user's info.

param

[in] Optional pointer to user data; will be passed unmodified to the callback function.

blocking

[in] If CHATTrue, return only after the operation has completed; otherwise, return immediately.

Remarks

The **chatGetBasicUserInfo** function is used to retrieve basic information on a user. This information consists of the chat nickname, user profile name, and IP address.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
chatGetBasicUserInfo	chatGetBasicUserInfoA	chatGetBasicUserInfoW

chatGetBasicUserInfoW and **chatGetBasicUserInfoA** are UNICODE and ANSI mapped versions of **chatGetBasicUserInfo**. The arguments of **chatGetBasicUserInfoA** are ANSI strings; those of **chatGetBasicUserInfoW** are wide-character strings.

Section Reference: [Gamespy Chat SDK](#)

See Also: [chatGetBasicUserInfoNoWait](#)

chatGetBasicUserInfoNoWait

Retrieves basic information on the specified user. Information is returned through function parameters.

```
CHATBool chatGetBasicUserInfoNoWait(  
    CHAT chat,  
    const gsi_char * nick,  
    const gsi_char ** user,  
    const gsi_char ** address );
```

Routine	Required Header	Distribution
chatGetBasicUserInfoNoWait	<chat.h>	SDKZIP

Return Value

Returns CHATTrue if info is available, CHATFalse otherwise.

Parameters

chat

[in] Chat SDK object, previously initialized using one of the chatConnect methods.

nick

[out] Receives the user's nickname

user

[out] Receives the user's username

address

[out] Receives the user's IP address.

Remarks

chatGetBasicUserInfoNoWait is used to retrieve basic information on a user. This information consists of the chat nickname, user profile name and IP address.

Section Reference: [Gamespy Chat SDK](#)

See Also: [chatGetBasicUserInfo](#)

chatGetChannelBasicUserInfo

Retrieves basic user info for every member of the specified channel.

```
void chatGetChannelBasicUserInfo(  
    CHAT chat,  
    const gsi_char * channel,  
    chatGetChannelBasicUserInfoCallback callback,  
    void * param,  
    CHATBool blocking );
```

Routine	Required Header	Distribution
chatGetChannelBasicUserInfo	<chat.h>	SDKZIP

Parameters

chat

[in] Chat SDK object, previously initialized using one of the chatConnect methods.

channel

[in] Name of the chat channel from which user information is being retrieved

callback

[in] Optional user-supplied function to be called when the operation has completed.

param

[in] Optional pointer to user data; will be passed unmodified to the callback function.

blocking

[in] If CHATTrue, return only after the operation has completed; otherwise, return immediately.

Remarks

The **chatGetChannelBasicUserInfo** function retrieves basic information for each of the users in the specified channel. The information returned consists of the nickname, profilename and IP address.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE
chatGetChannelBasicUserInfo	chatGetChannelBasicUserInfoA	chatGetChannelBasicUserInfoW

chatGetChannelBasicUserInfoW and **chatGetChannelBasicUserInfoA** are UNICODE and ANSI mapped versions of **chatGetChannelBasicUserInfo**. The arguments of **chatGetChannelBasicUserInfoA** are ANSI strings; those of **chatGetChannelBasicUserInfoW** are wide-character strings.

Section Reference: [Gamespy Chat SDK](#)

chatGetChannelKeys

Retrieves a list of key/value pairs for a channel or user.

```
void chatGetChannelKeys(  
    CHAT chat,  
    const gsi_char * channel,  
    const gsi_char * user,  
    int num,  
    const gsi_char ** keys,  
    chatGetChannelKeysCallback callback,  
    void * param,  
    CHATBool blocking );
```

Routine	Required Header	Distribution
chatGetChannelKeys	<chat.h>	SDKZIP

Parameters

chat

[in] Chat SDK object, previously initialized using one of the chatConnect methods.

channel

[in] Name of the chat channel from which key/value pairs are being retrieved

user

[in] Name of the user whose key/value pairs are being retrieved, or "*" to indicate the channel itself.

num

[in] Number of keys in the keys array.

keys

[in] Array of keys for which values will be returned.

callback

[in] Optional user-supplied function to be called when the operation has completed.

param

[in] Optional pointer to user data; will be passed unmodified to the callback function.

blocking

[in] If CHATTrue, return only after the operation has completed; otherwise, return immediately.

Remarks

The **chatGetChannelKeys** function retrieves a list of key/value pairs for the specified channel or user. If the user parameter is set to a user nickname, key/value pairs will be returned only for the specified user. If the user parameter is set to "*", values on the channel itself will be returned.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
chatGetChannelKeys	chatGetChannelKeysA	chatGetChannelKeysW

chatGetChannelKeysW and **chatGetChannelKeysA** are UNICODE and ANSI mapped versions of **chatGetChannelKeys**. The arguments of **chatGetChannelKeysA** are ANSI strings; those of **chatGetChannelKeysW** are wide-character strings.

Section Reference: [Gamespy Chat SDK](#)

chatGetChannelMode

Retrieves the "mode" of a channel.

```
void chatGetChannelMode(  
    CHAT chat,  
    const gsi_char * channel,  
    chatGetChannelModeCallback callback,  
    void * param,  
    CHATBool blocking );
```

Routine	Required Header	Distribution
chatGetChannelMode	<chat.h>	SDKZIP

Parameters

chat

[in] Chat SDK object, previously initialized using one of the chatConnect methods.

channel

[in] Name of the chat channel whose mode is being retrieved.

callback

[in] User-supplied function to receive mode information.

param

[in] Optional pointer to user data; will be passed unmodified to the callback function.

blocking

[in] If CHATTrue, return only after the operation has completed; otherwise, return immediately.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
chatGetChannelMode	chatGetChannelModeA	chatGetChannelModeW

chatGetChannelModeW and **chatGetChannelModeA** are UNICODE and ANSI mapped versions of **chatGetChannelMode**. The arguments of **chatGetChannelModeA** are ANSI strings; those of **chatGetChannelModeW** are wide-character strings.

Section Reference: [Gamespy Chat SDK](#)

See Also: [CHATChannelMode](#), [chatGetChannelModeCallback](#)

chatGetChannelNumUsers

Returns the number of users in the already joined channel. This is a cached value, and not a server query.

```
int chatGetChannelNumUsers(  
    CHAT chat,  
    const gsi_char * channel );
```

Routine	Required Header	Distribution
chatGetChannelNumUsers	<chat.h>	SDKZIP

Return Value

Returns the number of users in the channel. If the local client has not joined the channel, -1 will be returned.

Parameters

chat

[in] Chat SDK object, previously initialized using one of the chatConnect methods.

channel

[in] Name of the chat channel whose user count is being retrieved.

Section Reference: [Gamespy Chat SDK](#)

chatGetChannelPassword

Queries the server for the specified channel's password.

```
void chatGetChannelPassword(  
    CHAT chat,  
    const gsi_char * channel,  
    chatGetChannelPasswordCallback callback,  
    void * param,  
    CHATBool blocking );
```

Routine	Required Header	Distribution
chatGetChannelPassword	<chat.h>	SDKZIP

Parameters

chat

[in] Chat SDK object, previously initialized using one of the chatConnect methods.

channel

[in] Name of the chat channel whose password is being retrieved.

callback

[in] Optional user-supplied function to be called when the operation has completed.

param

[in] Optional pointer to user data; will be passed unmodified to the callback function.

blocking

[in] If CHATTrue, return only after the operation has completed; otherwise, return immediately.

Section Reference: [Gamespy Chat SDK](#)

See Also: [chatSetChannelPassword](#)

chatGetChannelTopic

Queries the server for the specified channel's topic. Also known as the room description.

```
void chatGetChannelTopic(  
    CHAT chat,  
    const gsi_char * channel,  
    chatGetChannelTopicCallback callback,  
    void * param,  
    CHATBool blocking );
```

Routine	Required Header	Distribution
chatGetChannelTopic	<chat.h>	SDKZIP

Parameters

chat

[in] Chat SDK object, previously initialized using one of the chatConnect methods.

channel

[in] Name of the chat channel whose topic is being retrieved.

callback

[in] Optional user-supplied function to be called when the operation has completed.

param

[in] Optional pointer to user data; will be passed unmodified to the callback function.

blocking

[in] If CHATTrue, return only after the operation has completed; otherwise, return immediately.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
chatGetChannelTopic	chatGetChannelTopicA	chatGetChannelTopicW

chatGetChannelTopicW and **chatGetChannelTopicA** are UNICODE and ANSI mapped versions of **chatGetChannelTopic**. The arguments of **chatGetChannelTopicA** are ANSI strings; those of **chatGetChannelTopicW** are wide-character strings.

Section Reference: [Gamespy Chat SDK](#)

See Also: [chatSetChannelTopic](#)

chatGetGlobalKeys

Retrieves a list of global keys for a single user, or all users.

```
void chatGetGlobalKeys(  
    CHAT chat,  
    const gsi_char * target,  
    int num,  
    const gsi_char ** keys,  
    chatGetGlobalKeysCallback callback,  
    void * param,  
    CHATBool blocking );
```

Routine	Required Header	Distribution
chatGetGlobalKeys	<chat.h>	SDKZIP

Parameters

chat

[in] Chat SDK object, previously initialized using one of the chatConnect methods.

target

[in] Target name, or NULL to specify all users.

num

[in] Number of keys to retrieve for each target.

keys

[in] Array of key names to request values for.

callback

[in] Optional user-supplied function to be called when the operation has completed.

param

[in] Optional pointer to user data; will be passed unmodified to the callback function.

blocking

[in] If CHATTrue, return only after the operation has completed; otherwise, return immediately.

Remarks

T.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
chatGetGlobalKeys	chatGetGlobalKeysA	chatGetGlobalKeysW

chatGetGlobalKeysW and **chatGetGlobalKeysA** are UNICODE and ANSI mapped versions of **chatGetGlobalKeys**. The arguments of **chatGetGlobalKeysA** are ANSI strings; those of **chatGetGlobalKeysW** are wide-character strings.

Section Reference: [Gamespy Chat SDK](#)

See Also: [chatSetGlobalKeys](#)

chatGetNick

Gets the chat nickname of the local client. This may not be the same as the profile nickname.

```
gsi_char * chatGetNick(  
    CHAT chat );
```

Routine	Required Header	Distribution
chatGetNick	<chat.h>	SDKZIP

Return Value

The nickname of the local client.

Parameters

chat

[in] Chat SDK object, previously initialized using one of the chatConnect methods.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
chatGetNick	chatGetNickA	chatGetNickW

chatGetNickW and **chatGetNickA** are UNICODE and ANSI mapped versions of **chatGetNick**. The arguments of **chatGetNickA** are ANSI strings; those of **chatGetNickW** are wide-character strings.

Section Reference: [Gamespy Chat SDK](#)

chatGetProfileID

Gets the profile id of the local client.

```
int chatGetProfileID(  
    CHAT chat );
```

Routine	Required Header	Distribution
chatGetProfileID	<chat.h>	SDKZIP

Return Value

Returns the profile id of the local client.

Parameters

chat

[in] Chat SDK object, previously initialized using `chatConnectLogin` or `chatConnectPreAuth`.

Remarks

The chat SDK must have been initialized using `chatConnectLogin` or `chatConnectPreAuth`.

Section Reference: [Gamespy Chat SDK](#)

See Also: [chatConnectLogin](#), [chatConnectPreAuth](#)

chatGetUserID

Gets the user id of the local client.

```
int chatGetUserID(  
    CHAT chat );
```

Routine	Required Header	Distribution
chatGetUserID	<chat.h>	SDKZIP

Return Value

Returns the user id of the local client.

Parameters

chat

[in] Chat SDK object, previously initialized using chatConnectLogin or chatConnectPreAuth.

Remarks

The chat SDK must have been initialized using `chatConnectLogin` or `chatConnectPreAuth`.

Section Reference: [Gamespy Chat SDK](#)

chatGetUserInfo

Gets information on the specified user.

```
void chatGetUserInfo(  
    CHAT chat,  
    const gsi_char * user,  
    chatGetUserInfoCallback callback,  
    void * param,  
    CHATBool blocking );
```

Routine	Required Header	Distribution
chatGetUserInfo	<chat.h>	SDKZIP

Parameters

chat

[in] Chat SDK object, previously initialized using one of the chatConnect methods.

user

[in] User's chat nickname.

callback

[in] Optional user-supplied function to be called when the operation has completed.

param

[in] Optional pointer to user data; will be passed unmodified to the callback function.

blocking

[in] If CHATTrue, return only after the operation has completed; otherwise, return immediately.

Remarks

The user information includes the user's profile nickname, username, real name and address. The callback also contains the channels that this user is a member of.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
chatGetUserInfo	chatGetUserInfoA	chatGetUserInfoW

chatGetUserInfoW and **chatGetUserInfoA** are UNICODE and ANSI mapped versions of **chatGetUserInfo**. The arguments of **chatGetUserInfoA** are ANSI strings; those of **chatGetUserInfoW** are wide-character strings.

Section Reference: [Gamespy Chat SDK](#)

chatGetUserMode

Get the mode of a user in a specified channel.

```
void chatGetUserMode(  
    CHAT chat,  
    const gsi_char * channel,  
    const gsi_char * user,  
    chatGetUserModeCallback callback,  
    void * param,  
    CHATBool blocking );
```

Routine	Required Header	Distribution
chatGetUserMode	<chat.h>	SDKZIP

Parameters

chat

[in] Chat SDK object, previously initialized using one of the chatConnect methods.

channel

[in] Name of the chat channel being inspected.

user

[in] User's chat nickname on that channel.

callback

[in] Optional user-supplied function to be called when the operation has completed; will be passed user's mode.

param

[in] Optional pointer to user data; will be passed unmodified to the callback function.

blocking

[in] If CHATTrue, return only after the operation has completed; otherwise, return immediately.

Remarks

The **chatGetUserMode** function may be used to check a user's "mode" in a specified chat channel. A mode may specify a channel operator.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
chatGetUserMode	chatGetUserModeA	chatGetUserModeW

chatGetUserModeW and **chatGetUserModeA** are UNICODE and ANSI mapped versions of **chatGetUserMode**. The arguments of **chatGetUserModeA** are ANSI strings; those of **chatGetUserModeW** are wide-character strings.

Section Reference: [Gamespy Chat SDK](#)

See Also: [chatGetUserModeNoWait](#)

chatGetUserModeNoWait

Get the mode of a user in a specified channel, returning it through a function parameter.

```
CHATBool chatGetUserModeNoWait(  
    CHAT chat,  
    const gsi_char * channel,  
    const gsi_char * user,  
    int * mode );
```

Routine	Required Header	Distribution
chatGetUserModeNoWait	<chat.h>	SDKZIP

Return Value

Parameters

chat

[in] Chat SDK object, previously initialized using one of the chatConnect methods.

channel

[in] Name of the chat channel being inspected.

user

[in] User's chat nickname on that channel.

mode

[out] Receives the mode of target user.

Remarks

The **chatGetUserModeNoWait** function may be used to check a user's "mode" in a specified chat channel. A mode may specify a channel operator.

Section Reference: [Gamespy Chat SDK](#)

See Also: [chatGetUserMode](#)

chatInChannel

Determine whether the local client is a member of the specified channel.

```
CHATBool chatInChannel(  
    CHAT chat,  
    const gsi_char * channel );
```

Routine	Required Header	Distribution
chatInChannel	<chat.h>	SDKZIP

Return Value

This function will return CHATTrue if the local client is a member of the specified channel, CHATFalse otherwise.

Parameters

chat

[in] Chat SDK object, previously initialized using one of the chatConnect methods.

channel

[in] Name of the chat channel being inspected.

Remarks

The **chatInChannel** function checks the local list of channels to determine whether the local client is a member. No communication with the server is attempted during this call.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
chatInChannel	chatInChannelA	chatInChannelW

chatInChannelW and **chatInChannelA** are UNICODE and ANSI mapped versions of **chatInChannel**. The arguments of **chatInChannelA** are ANSI strings; those of **chatInChannelW** are wide-character strings.

Section Reference: [Gamespy Chat SDK](#)

chatInviteUser

Invite a user to join a channel.

```
void chatInviteUser(  
    CHAT chat,  
    const gsi_char * channel,  
    const gsi_char * user );
```

Routine	Required Header	Distribution
chatInviteUser	<chat.h>	SDKZIP

Parameters

chat

[in] Chat SDK object, previously initialized using one of the chatConnect methods.

channel

[in] Name of the chat channel to which the user is being invited.

user

[in] User's chat nickname.

Remarks

The **chatInviteUser** function may be used to invite a user to a particular chat room.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
chatInviteUser	chatInviteUserA	chatInviteUserW

chatInviteUserW and **chatInviteUserA** are UNICODE and ANSI mapped versions of **chatInviteUser**. The arguments of **chatInviteUserA** are ANSI strings; those of **chatInviteUserW** are wide-character strings.

Section Reference: [Gamespy Chat SDK](#)

chatKickUser

Forcefully remove a user from a specified channel. .

```
void chatKickUser(  
    CHAT chat,  
    const gsi_char * channel,  
    const gsi_char * user,  
    const gsi_char * reason );
```

Routine	Required Header	Distribution
chatKickUser	<chat.h>	SDKZIP

Parameters

chat

[in] Chat SDK object, previously initialized using one of the chatConnect methods.

channel

[in] Name of the chat channel from which the user is being removed.

user

[in] User's chat nickname.

reason

[in] Optional text string that will be sent along with the kick message. This message will appear in the user kick callback.

Remarks

The local client must have operator privileges to execute this command.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
chatKickUser	chatKickUserA	chatKickUserW

chatKickUserW and **chatKickUserA** are UNICODE and ANSI mapped versions of **chatKickUser**. The arguments of **chatKickUserA** are ANSI strings; those of **chatKickUserW** are wide-character strings.

Section Reference: [Gamespy Chat SDK](#)

chatLeaveChannel

Leave a chat channel.

```
void chatLeaveChannel(  
    CHAT chat,  
    const gsi_char * channel,  
    const gsi_char * reason );
```

Routine	Required Header	Distribution
chatLeaveChannel	<chat.h>	SDKZIP

Parameters

chat

[in] Chat SDK object, previously initialized using one of the chatConnect methods.

channel

[in] Name of the chat channel being left.

reason

[in] Optional reason for leaving. This may be displayed to the remaining users.

Remarks

The **chatLeaveChannel** function is used to remove the local client from a chat channel.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
chatLeaveChannel	chatLeaveChannelA	chatLeaveChannelW

chatLeaveChannelW and **chatLeaveChannelA** are UNICODE and ANSI mapped versions of **chatLeaveChannel**. The arguments of **chatLeaveChannelA** are ANSI strings; those of **chatLeaveChannelW** are wide-character strings.

Section Reference: [Gamespy Chat SDK](#)

See Also: [ChatConnect](#), [ChatDisconnect](#)

chatRegisterUniqueNick

Registers a unique nick to the local client and cdkey.

```
void chatRegisterUniqueNick(  
    CHAT chat,  
    int namespaceID,  
    const gsi_char * uniquenick,  
    const gsi_char * cdkey );
```

Routine	Required Header	Distribution
chatRegisterUniqueNick	<chat.h>	SDKZIP

Parameters

chat

[in] Chat SDK object, previously initialized using one of the chatConnect methods.

namespaceID

[in] ID of the namespace community. Assigned by GameSpy.

uniquenick

[in] Nickname being registered.

cdkey

[in] User's CD key; this uniquely identifies the account.

Remarks

The **chatRegisterUniqueNick** function should be used in response to a `chatNickErrorCallback`. This function requests that a specified unique nick be associated with the local client and `cdkey`. If an error occurs, another `chatNickErrorCallback` will be triggered. Take care that this does not result in an infinite loop.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
chatRegisterUniqueNick	chatRegisterUniqueNickA	chatRegisterUniqueNi

chatRegisterUniqueNickW and **chatRegisterUniqueNickA** are UNICODE and ANSI mapped versions of **chatRegisterUniqueNick**. The arguments of **chatRegisterUniqueNickA** are ANSI strings; those of **chatRegisterUniqueNickW** are wide-character strings.

Section Reference: [Gamespy Chat SDK](#)

chatRemoveChannelBan

Removes a banned player from a channel's ban list. This will once again allow the user to join the channel.

```
void chatRemoveChannelBan(  
    CHAT chat,  
    const gsi_char * channel,  
    const gsi_char * ban );
```

Routine	Required Header	Distribution
chatRemoveChannelBan	<chat.h>	SDKZIP

Parameters

chat

[in] Chat SDK object, previously initialized using one of the chatConnect methods.

channel

[in] Name of the chat channel whose ban list is being modified..

ban

[in] Nickname to remove from the ban list.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defi
chatRemoveChannelBan	chatRemoveChannelBanA	chatRemoveChanne

chatRemoveChannelBanW and **chatRemoveChannelBanA** are UNICODE and ANSI mapped versions of **chatRemoveChannelBan**. The arguments of **chatRemoveChannelBanA** are ANSI strings; those of **chatRemoveChannelBanW** are wide-character strings.

Section Reference: [Gamespy Chat SDK](#)

chatRetryWithNick

Use in response to a nickErrorCallback. This function allows the local client to retry the connection attempt with a different chat nickname.

```
void chatRetryWithNick(  
    CHAT chat,  
    const gsi_char * nick );
```

Routine	Required Header	Distribution
chatRetryWithNick	<chat.h>	SDKZIP

Parameters

chat

[in] Chat SDK object, previously initialized using one of the chatConnect methods.

nick

[in] Alternate chat nickname

Remarks

The **chatRetryWithNick** function should be used in response to a `nickErrorCallback`. Most often, this occurs when a requested nickname is already in use. **chatRetryWithNick** should be called with an alternate nickname such as "oldnick{1}" to continue the login process. If another `nickError` occurs, the `nickErrorCallback` will be triggered again.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
chatRetryWithNick	chatRetryWithNickA	chatRetryWithNickW

chatRetryWithNickW and **chatRetryWithNickA** are UNICODE and ANSI mapped versions of **chatRetryWithNick**. The arguments of **chatRetryWithNickA** are ANSI strings; those of **chatRetryWithNickW** are wide-character strings.

Section Reference: [Gamespy Chat SDK](#)

chatSendChannelMessage

Send a message to all members of the specified channel.

```
void chatSendChannelMessage(  
    CHAT chat,  
    const gsi_char * channel,  
    const gsi_char * message,  
    int type );
```

Routine	Required Header	Distribution
chatSendChannelMessage	<chat.h>	SDKZIP

Parameters

chat

[in] Chat SDK object, previously initialized using one of the chatConnect methods.

channel

[in] Name of the chat channel to which the message is being sent.

message

[in] Message.

type

[in] One of the predefined chat types. Used to send chat, hidden messages, notices, and other types.

Remarks

The **chatSendChannelMessage** function is used to send a message to all users of a specified channel. The type of message that may be sent can be chat, UTM, notices or actions.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE
chatSendChannelMessage	chatSendChannelMessageA	chatSendChann

chatSendChannelMessageW and **chatSendChannelMessageA** are UNICODE and ANSI mapped versions of **chatSendChannelMessage**. The arguments of **chatSendChannelMessageA** are ANSI strings; those of **chatSendChannelMessageW** are wide-character strings.

Section Reference: [Gamespy Chat SDK](#)

chatSendRaw

Send a raw command to the chat server. This does not automatically send to a player.

```
void chatSendRaw(  
    CHAT chat,  
    const gsi_char * command );
```

Routine	Required Header	Distribution
chatSendRaw	<chat.h>	SDKZIP

Parameters

chat

[in] Chat SDK object, previously initialized using one of the chatConnect methods.

command

[in] Raw command to send to the chat server.

Remarks

The **chatSendRaw** function may be used to send a raw command to the server. Special care should be taken when using this command, as undesired behavior may result from malformed command sequences. If in doubt, please contact developer support on the use of this command.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
chatSendRaw	chatSendRawA	chatSendRawW

chatSendRawW and **chatSendRawA** are UNICODE and ANSI mapped versions of **chatSendRaw**. The arguments of **chatSendRawA** are ANSI strings; those of **chatSendRawW** are wide-character strings.

Section Reference: [Gamespy Chat SDK](#)

chatSendUserMessage

Send a private message to a user.

```
void chatSendUserMessage(  
    CHAT chat,  
    const gsi_char * user,  
    const gsi_char * message,  
    int type );
```

Routine	Required Header	Distribution
chatSendUserMessage	<chat.h>	SDKZIP

Parameters

chat

[in] Chat SDK object, previously initialized using one of the chatConnect methods.

user

[in] Nickname of the user to whom the private message is being sent.

message

[in] Message; generally chat text, but may also be a raw data message.

type

[in] One of the ChatType predefined types; can signify a chat message or a raw data message.

Remarks

The **chatSendUserMessage** function to send a private message to a specified user. The recipient does not need to be in the same room as the sender.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
chatSendUserMessage	chatSendUserMessageA	chatSendUserMessageW

chatSendUserMessageW and **chatSendUserMessageA** are UNICODE and ANSI mapped versions of **chatSendUserMessage**. The arguments of **chatSendUserMessageA** are ANSI strings; those of **chatSendUserMessageW** are wide-character strings.

Section Reference: [Gamespy Chat SDK](#)

chatSetChannelGroup

Assign a user-defined grouping to a channel. The group is a string identifier which is linked to the channel.

```
void chatSetChannelGroup(  
    CHAT chat,  
    const gsi_char * channel,  
    const gsi_char * group );
```

Routine	Required Header	Distribution
chatSetChannelGroup	<chat.h>	SDKZIP

Parameters

chat

[in] Chat SDK object, previously initialized using one of the chatConnect methods.

channel

[in] Name of the chat channel to which a group is being assigned.

group

[in] Group string to assign to channel.

Remarks

The **chatSetChannelGroup** function may be used to attach a user-defined string to a channel. This string exists locally and is not sent across the network. This string may be used as a local grouping for channels.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
chatSetChannelGroup	chatSetChannelGroupA	chatSetChannelGroupW

chatSetChannelGroupW and **chatSetChannelGroupA** are UNICODE and ANSI mapped versions of **chatSetChannelGroup**. The arguments of **chatSetChannelGroupA** are ANSI strings; those of **chatSetChannelGroupW** are wide-character strings.

Section Reference: [Gamespy Chat SDK](#)

chatSetChannelKeys

Set key/values on a channel or the local user.

```
void chatSetChannelKeys(  
    CHAT chat,  
    const gsi_char * channel,  
    const gsi_char * user,  
    int num,  
    const gsi_char ** keys,  
    const gsi_char ** values );
```

Routine	Required Header	Distribution
chatSetChannelKeys	<chat.h>	SDKZIP

Parameters

chat

[in] Chat SDK object, previously initialized using one of the chatConnect methods.

channel

[in] Name of the chat channel whose keys are being set.

user

[in] User to assign keys to. May be NULL. Only channel operators may set keys on other players.

num

[in] Number of key/value pairs being set.

keys

[in] Array of keys being set.

values

[in] Array of values being set, in the same order as their keys.

Remarks

The **chatSetChannelKeys** function may be used to set channel keys on a member or on the channel itself. Only channel operators may set keys on other players.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
chatSetChannelKeys	chatSetChannelKeysA	chatSetChannelKeysW

chatSetChannelKeysW and **chatSetChannelKeysA** are UNICODE and ANSI mapped versions of **chatSetChannelKeys**. The arguments of **chatSetChannelKeysA** are ANSI strings; those of **chatSetChannelKeysW** are wide-character strings.

Section Reference: [Gamespy Chat SDK](#)

chatSetChannelLimit

Set the maximum number of users allowed in a channel.

```
void chatSetChannelLimit(  
    CHAT chat,  
    const gsi_char * channel,  
    int limit );
```

Routine	Required Header	Distribution
chatSetChannelLimit	<chat.h>	SDKZIP

Parameters

chat

[in] Chat SDK object, previously initialized using one of the chatConnect methods.

channel

[in] Name of the chat channel whose limit is being set.

limit

[in] Maximum number of users on channel.

Remarks

The **chatSetChannelLimit** function may be used to set the maximum number of users on a chat room.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
chatSetChannelLimit	chatSetChannelLimitA	chatSetChannelLimitW

chatSetChannelLimitW and **chatSetChannelLimitA** are UNICODE and ANSI mapped versions of **chatSetChannelLimit**. The arguments of **chatSetChannelLimitA** are ANSI strings; those of **chatSetChannelLimitW** are wide-character strings.

Section Reference: [Gamespy Chat SDK](#)

chatSetChannelMode

Set a channel's mode.

```
void chatSetChannelMode(  
    CHAT chat,  
    const gsi_char * channel,  
    CHATChannelMode * mode );
```

Routine	Required Header	Distribution
chatSetChannelMode	<chat.h>	SDKZIP

Parameters

chat

[in] Chat SDK object, previously initialized using one of the chatConnect methods.

channel

[in] Name of the chat channel whose mode is being set.

mode

[in] Properties to set on the target channel.

Remarks

The mode includes standard IRC properties such as "InviteOnly, Private and Moderated".

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
chatSetChannelMode	chatSetChannelModeA	chatSetChannelModeW

chatSetChannelModeW and **chatSetChannelModeA** are UNICODE and ANSI mapped versions of **chatSetChannelMode**. The arguments of **chatSetChannelModeA** are ANSI strings; those of **chatSetChannelModeW** are wide-character strings.

Section Reference: [Gamespy Chat SDK](#)

See Also: [CHATChannelMode](#), [chatGetChannelMode](#)

chatSetChannelPassword

Sets or clears a password on the specified channel.

```
void chatSetChannelPassword(  
    CHAT chat,  
    const gsi_char * channel,  
    CHATBool enable,  
    const gsi_char * password );
```

Routine	Required Header	Distribution
chatSetChannelPassword	<chat.h>	SDKZIP

Parameters

chat

[in] Chat SDK object, previously initialized using one of the chatConnect methods.

channel

[in] Name of the chat channel whose password is being set.

enable

[in] If CHATTrue, enable the password; otherwise, disable.

password

[in] Password string which users must supply to join the channel.

Remarks

Set the value to NULL or "" to clear the value.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE De
chatSetChannelPassword	chatSetChannelPasswordA	chatSetChannelPa

chatSetChannelPasswordW and **chatSetChannelPasswordA** are UNICODE and ANSI mapped versions of **chatSetChannelPassword**. The arguments of **chatSetChannelPasswordA** are ANSI strings; those of **chatSetChannelPasswordW** are wide-character strings.

Section Reference: [Gamespy Chat SDK](#)

chatSetChannelTopic

Set the topic (description) of a chat channel.

```
void chatSetChannelTopic(  
    CHAT chat,  
    const gsi_char * channel,  
    const gsi_char * topic );
```

Routine	Required Header	Distribution
chatSetChannelTopic	<chat.h>	SDKZIP

Parameters

chat

[in] Chat SDK object, previously initialized using one of the chatConnect methods.

channel

[in] Name of the chat channel whose topic is being set.

topic

[in] Description of new topic.

Remarks

The **chatSetChannelTopic** function is used to set the topic (description) of a chat channel. Some channels, such as the title and group rooms, will not allow users to set the topic.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
chatSetChannelTopic	chatSetChannelTopicA	chatSetChannelTopicW

chatSetChannelTopicW and **chatSetChannelTopicA** are UNICODE and ANSI mapped versions of **chatSetChannelTopic**. The arguments of **chatSetChannelTopicA** are ANSI strings; those of **chatSetChannelTopicW** are wide-character strings.

Section Reference: [Gamespy Chat SDK](#)

See Also: [chatGetChannelTopic](#)

chatSetGlobalKeys

Set key/values on the local client. .

```
void chatSetGlobalKeys(  
    CHAT chat,  
    int num,  
    const gsi_char ** keys,  
    const gsi_char ** values );
```

Routine	Required Header	Distribution
chatSetGlobalKeys	<chat.h>	SDKZIP

Parameters

chat

[in] Chat SDK object, previously initialized using one of the chatConnect methods.

num

[in] Number of key/value pairs being set.

keys

[in] Array of keys being set.

values

[in] Array of values being set, in the same order as their keys.

Remarks

Set the value to NULL or "" to clear the value.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
chatSetGlobalKeys	chatSetGlobalKeysA	chatSetGlobalKeysW

chatSetGlobalKeysW and **chatSetGlobalKeysA** are UNICODE and ANSI mapped versions of **chatSetGlobalKeys**. The arguments of **chatSetGlobalKeysA** are ANSI strings; those of **chatSetGlobalKeysW** are wide-character strings.

Section Reference: [Gamespy Chat SDK](#)

chatSetQuietMode

Sets the chat sdk to quiet mode or disables quiet mode.

```
void chatSetQuietMode(  
    CHAT chat,  
    CHATBool quiet );
```

Routine	Required Header	Distribution
chatSetQuietMode	<chat.h>	SDKZIP

Parameters

chat

[in] Chat SDK object, previously initialized using one of the chatConnect methods.

quiet

[in] If CHATTrue, enable quiet mode; otherwise, disable.

Remarks

The **chatSetQuietMode** function is used to toggle quiet mode. When in quiet mode the chat SDK will not receive chat or other messages. This allows the user to remain logged into chat without disrupting gameplay with extraneous traffic.

Section Reference: [Gamespy Chat SDK](#)

chatSetUserMode

Set the IRC mode of the specified user. This mode is applied in the specified channel.

```
void chatSetUserMode(  
    CHAT chat,  
    const gsi_char * channel,  
    const gsi_char * user,  
    int mode );
```

Routine	Required Header	Distribution
chatSetUserMode	<chat.h>	SDKZIP

Parameters

chat

[in] Chat SDK object, previously initialized using one of the chatConnect methods.

channel

[in] Name of the user's chat channel.

user

[in] User's chat nickname on that channel.

mode

[in] User mode flags. See Remarks.

Remarks

The **chatSetUserMode** function may be used to set a user's mode in a particular channel. Modes are used to track which users have operator and speaking privileges.

The following user mode flags are defined:

CHAT_NORMAL -- Normal (no speaking privileges; no operator privileges)

CHAT_VOICE -- User has speaking privileges.

CHAT_OP -- User has operator privileges.

User mode flags may be OR'ed together. CHAT_NORMAL is superseded by any other user mode flag.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
chatSetUserMode	chatSetUserModeA	chatSetUserModeW

chatSetUserModeW and **chatSetUserModeA** are UNICODE and ANSI mapped versions of **chatSetUserMode**. The arguments of **chatSetUserModeA** are ANSI strings; those of **chatSetUserModeW** are wide-character strings.

Section Reference: [Gamespy Chat SDK](#)

chatThink

Allow the Chat SDK to continue processing.

```
void chatThink(  
    CHAT chat );
```

Routine	Required Header	Distribution
chatThink	<chat.h>	SDKZIP

Parameters

chat

[in] Chat SDK object, previously initialized using one of the chatConnect methods.

Remarks

All network communications, callbacks and other events will happen only during this call. The frequency with which this method is called will affect general performance on the SDK.

Section Reference: [Gamespy Chat SDK](#)

See Also: [ChatConnect](#), [ChatDisconnect](#)

chatTranslateNick

Removes the namespace extension from a nickname. Use this when working with unique nicknames in a public chat room.

```
const gsi_char * chatTranslateNick(  
    gsi_char * nick,  
    const gsi_char * extension );
```

Routine	Required Header	Distribution
chatTranslateNick	<chat.h>	SDKZIP

Return Value

Returns the nickname, stripped of the namespace identifier.

Parameters

nick

[in] Current nickname.

extension

[in] Game extension; will be removed from the nickname. Assigned by GameSpy.

Remarks

Nicknames that are registered in a game's namespace will include an identifying extension, such as "-gspy". This extension should not be displayed to the user, but should be stripped before display.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
chatTranslateNick	chatTranslateNickA	chatTranslateNickW

chatTranslateNickW and **chatTranslateNickA** are UNICODE and ANSI mapped versions of **chatTranslateNick**. The arguments of **chatTranslateNickA** are ANSI strings; those of **chatTranslateNickW** are wide-character strings.

Section Reference: [Gamespy Chat SDK](#)

Chat SDK Callbacks

chatAuthenticateCDKeyCallback	Called when chatAuthenticateCDKey and attempt to authenticate the CD-Key is finished.
chatBroadcastKeyChanged	Called when a player changes a broadcast key in a channel the local player is in
chatChangeNickCallback	Callback for chatChangeNick when a player changes his/her nick.
chatChannelMessage	Used in conjunction with chatEnterChannel; called when a message is received in the channel.
chatChannelModeChanged	Used in conjunction with chatEnterChannel; called when the mode of a user in the channel changes.
chatConnectCallback	Called when a chatConnect* attempt is made
chatDisconnected	Called when a disconnection occurs.

chatEnterChannelCallback	Called when an attempt to enter the channel has completed
chatEnumChannelBansCallback	Called after an attempt to enumerate channel bans.
chatEnumChannelsCallbackAll	Called when an attempt to enumerate all the channels is complete
chatEnumChannelsCallbackEach	Called after an attempt to enumerate each channel.
chatEnumJoinedChannelsCallback	Called after an attempt to enumerate joined channels.
chatEnumUsersCallback	Called after an attempt to enumerate the users in a channel
chatFillInUserCallback	Used in conjunction with the chatConnectSpecial and chatConnectSecure functions; called to fill in the user field after the actual network connection to the chat server has been made.
chatGetBasicUserInfoCallback	Called after an attempt to get basic information on a user
chatGetChannelBasicUserInfoCallback	Called when an attempt to get everyone's basic user info is made.

chatGetChannelKeysCallback	Called after an attempt to get the channel keys or user(s) keys
chatGetChannelModeCallback	Called after an attempt to get the channel mode.
chatGetChannelPasswordCallback	Called after an attempt to get the channel's password.
chatGetChannelTopicCallback	Called after an attempt to get the channel's topic.
chatGetGlobalKeysCallback	Called after an attempt to get the global keys for the user(s).
chatGetUserInfoCallback	Called after an attempt to get user information.
chatGetUserModeCallback	Called after an attempt to get the user's mode
chatInvited	Used in conjunction with the chatConnect functions; called when the local user gets invited to a channel.
chatKicked	Used in conjunction with chatEnterChannel; called when the local user gets kicked from the channel.

chatNewUserList	Used in conjunction with chatEnterChannel; Called when the chat server sends an entire new user list for a channel we're in.
chatNickErrorCallback	Used in conjunction with the chatConnect functions; called if there was an error with the provided nickname.
chatPrivateMessage	Used in conjunction with the chatConnect functions; called when a message is received from another user.
chatRaw	Used in conjunction with the chatConnect functions; all raw incoming network traffic gets passed to this function.
chatTopicChanged	Used in conjunction with chatEnterChannel; called when the channel topic changes.
chatUserChangedNick	Used in conjunction with chatEnterChannel; called when a user in the channel changes their nickname.
chatUserJoined	Used in conjunction with

	chatEnterChannel; called when a user joins the channel.
chatUserListUpdated	Used in conjunction with chatEnterChannel; called when the channel's user list changes.
chatUserModeChanged	Used in conjunction with chatEnterChannel; called when the mode of a user in the channel changes.
chatUserParted	Used in conjunction with chatEnterChannel; called when a user parts the channel.

chatAuthenticateCDKeyCallback

Called when chatAuthenticateCDKey and attempt to authenticate the CD-Key is finished.

```
typedef void (*chatAuthenticateCDKeyCallback)(  
    CHAT chat,  
    int result,  
    const gsi_char * message,  
    void * param );
```

Routine	Required Header	Distribution
chatAuthenticateCDKeyCallback	<chat.h>	SDKZIP

Parameters

chat

[in] The initialized chat interface object.

result

[in] Indicates the result of the attempt.

message

[in] The text message representing the result.

param

[in] Pointer to user data. Passed through unmodified from the initiating function.

Remarks

The **chatAuthenticateCDKeyCallback** function gets called when an attempt to authenticate a CD key is finished. If the result has a value of 1, the CD key was authenticated. Otherwise, the CD key was not authenticated.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_
chatAuthenticateCDKeyCallback	chatAuthenticateCDKeyCallbackA	chatA

chatAuthenticateCDKeyCallbackW and **chatAuthenticateCDKeyCallbackA** are UNICODE and ANSI mapped versions of **chatAuthenticateCDKeyCallback**. The arguments of **chatAuthenticateCDKeyCallbackA** are ANSI strings; those of **chatAuthenticateCDKeyCallbackW** are wide-character strings.

Section Reference: [Gamespy Chat SDK](#)

chatBroadcastKeyChanged

Called when a player changes a broadcast key in a channel the local player is in.

```
typedef void (*chatBroadcastKeyChanged)(  
    CHAT chat,  
    const gsi_char *channel,  
    const gsi_char *user,  
    const gsi_char *key,  
    const gsi_char *value,  
    void *param );
```

Routine	Required Header	Distribution
chatBroadcastKeyChanged	<chat.h>	SDKZIP

Parameters

chat

[in] The initialized chat interface object.

channel

[in] The channel the local player is in.

user

[in] The nickname of the user who changed the key

key

[in] The broadcast key that was changed

value

[in] The broadcast key value

param

[in] Pointer to user data. The same param that was passed to chatEnterChannel through the callbacks structure.

Remarks

The **chatBroadcastKeyChanged** function is called when another player changes a broadcast key in the channel the local player is in.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE
chatBroadcastKeyChanged	chatBroadcastKeyChangedA	chatBroadcastK

chatBroadcastKeyChangedW and **chatBroadcastKeyChangedA** are UNICODE and ANSI mapped versions of **chatBroadcastKeyChanged**. The arguments of **chatBroadcastKeyChangedA** are ANSI strings; those of **chatBroadcastKeyChangedW** are wide-character strings.

Section Reference: [Gamespy Chat SDK](#)

chatChangeNickCallback

Callback for chatChangeNick when a player changes his/her nick.

```
typedef void (*chatChangeNickCallback)(  
    CHAT chat,  
    CHATBool success,  
    const gsi_char * oldNick,  
    const gsi_char * newNick,  
    void * param );
```

Routine	Required Header	Distribution
chatChangeNickCallback	<chat.h>	SDKZIP

Parameters

chat

[in] The initialized chat interface object.

success

[in] CHATTrue if success, CHATFalse if failure.

oldNick

[in] The old nickname.

newNick

[in] The new nickname.

param

[in] User data; the same param pointer that was passed to chatChangeNick.

Remarks

The chatChangedNickCallback is called when any player in the specified room changes his/her nick. The new nick is assigned to the player if the change was validated by the server. Otherwise, there will be no difference between the old nick or the new nick. The change is determined by "success" which is either CHATTrue or CHATFalse.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Def
chatChangeNickCallback	chatChangeNickCallbackA	chatChangeNickCal

chatChangeNickCallbackW and **chatChangeNickCallbackA** are UNICODE and ANSI mapped versions of **chatChangeNickCallback**. The arguments of **chatChangeNickCallbackA** are ANSI strings; those of **chatChangeNickCallbackW** are wide-character strings.

Section Reference: [Gamespy Chat SDK](#)

chatChannelMessage

Used in conjunction with chatEnterChannel; called when a message is received in the channel.

```
typedef void (*chatChannelMessage)(  
    CHAT chat,  
    const gsi_char *channel,  
    const gsi_char *user,  
    const gsi_char *message,  
    int type,  
    void *param );
```

Routine	Required Header	Distribution
chatChannelMessage	<chat.h>	SDKZIP

Parameters

chat

[in] The initialized chat interface object.

channel

[in] The channel the local player is in.

user

[in] The nickname of the user who sent the message.

message

[in] The text of the message.

type

[in] The type of the message: one of the pre-defined chat types.

param

[in] Pointer to user data. The same param that was passed to chatEnterChannel through the callbacks structure.

Section Reference: [Gamespy Chat SDK](#)

See Also: [chatEnterChannel](#), [chatSendChannelMessage](#)

chatChannelModeChanged

Used in conjunction with chatEnterChannel; called when the mode of a user in the channel changes.

```
typedef void (*chatChannelModeChanged)(  
    CHAT chat,  
    const gsi_char *channel,  
    CHATChannelMode *mode,  
    void *param );
```

Routine	Required Header	Distribution
chatChannelModeChanged	<chat.h>	SDKZIP

Parameters

chat

[in] The initialized chat interface object.

channel

[in] The channel the local player is in.

mode

[in] Properties of the new mode set on the channel.

param

[in] Pointer to user data. The same param that was passed to chatEnterChannel through the callbacks structure.

Section Reference: [Gamespy Chat SDK](#)

See Also: [chatEnterChannel](#), [chatSetChannelMode](#)

chatConnectCallback

Called when a chatConnect* attempt is made.

```
typedef void (*chatConnectCallback)(  
    CHAT chat,  
    CHATBool success,  
    int failureReason,  
    void * param );
```

Routine	Required Header	Distribution
chatConnectCallback	<chat.h>	SDKZIP

Parameters

chat

[in] The initialized chat interface object.

success

[in] CHATTrue if success, CHATFalse if failure.

failureReason

[in] The string giving reason for failure

param

[in] Pointer to user data. Passed through unmodified from the initiating function.

Remarks

The **chatConnectCallback** is called after an attempt of a call to one of the connect functions that the Chat SDK provides.

Section Reference: [Gamespy Chat SDK](#)

chatDisconnected

Called when a disconnection occurs.

```
typedef void (*chatDisconnected)(  
    CHAT chat,  
    const gsi_char * reason,  
    void * param );
```

Routine	Required Header	Distribution
chatDisconnected	<chat.h>	SDKZIP

Parameters

chat

[in] The initialized chat interface object.

reason

[in] The text string which states the reason for disconnect

param

[in] Pointer to user data. The same param that was passed to chatConnect through the callback structure.

Remarks

The **chatDisconnected** callback function is called after a disconnection occurs. The connection can be ended at any time by called `chatDisconnect()`. If the connection gets disconnected for any other reason (such as an intermediate router going down), the **chatDisconnected()** callback will be called.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
chatDisconnected	chatDisconnectedA	chatDisconnectedW

chatDisconnectedW and **chatDisconnectedA** are UNICODE and ANSI mapped versions of **chatDisconnected**. The arguments of **chatDisconnectedA** are ANSI strings; those of **chatDisconnectedW** are wide-character strings.

Section Reference: [Gamespy Chat SDK](#)

chatEnterChannelCallback

Called when an attempt to enter the channel has completed.

```
typedef void (*chatEnterChannelCallback)(  
    CHAT chat,  
    CHATBool success,  
    CHATEnterResult result,  
    const gsi_char * channel,  
    void * param );
```

Routine	Required Header	Distribution
chatEnterChannelCallback	<chat.h>	SDKZIP

Parameters

chat

[in] The initialized chat interface object.

success

[in] CHATTrue if success, CHATFalse if failure.

result

[in] Indicates the result of the attempt

channel

[in] The name of channel entered

param

[in] Pointer to user data. Passed through unmodified from the initiating function.

Remarks

The **chatEnterChannelCallback** function is called when the attempt to enter the channel by the local player is completed. The entrance of the channel can be successful or a failure, and is indicated by the "result" of the attempt. The "result" can be of the following value:

CHATEnterSuccess -- The channel was successfully entered.

CHATBadChannelName -- The channel name was invalid.

CHATChannellsFull -- The channel is at its user limit.

CHATInviteOnlyChannel -- The channel is invite only.

CHATBannedFromChannel -- The local user is banned from this channel.

CHATBadChannelPassword -- The channel has a password, and a bad password (or none) was given.

CHATTooManyChannels -- The server won't allow this user in any more channels.

CHATEnterTimedOut -- The attempt to enter timed out.

CHATBadChannelMask -- The channel mask was bad (rarely used).

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE D
chatEnterChannelCallback	chatEnterChannelCallbackA	chatEnterChanne

chatEnterChannelCallbackW and **chatEnterChannelCallbackA** are UNICODE and ANSI mapped versions of **chatEnterChannelCallback**. The arguments of **chatEnterChannelCallbackA** are ANSI strings; those of **chatEnterChannelCallbackW** are wide-character strings.

Section Reference: [Gamespy Chat SDK](#)

chatEnumChannelBansCallback

Called after an attempt to enumerate channel bans.

```
typedef void (*chatEnumChannelBansCallback)(  
    CHAT chat,  
    CHATBool success,  
    const gsi_char * channel,  
    int numBans,  
    const gsi_char ** bans,  
    void * param );
```

Routine	Required Header	Distribution
chatEnumChannelBansCallback	<chat.h>	SDKZIP

Parameters

chat

[in] The initialized chat interface object.

success

[in] CHATTrue if success, CHATFalse if failure.

channel

[in] The channel that the enumeration was attempted

numBans

[in] The number of bans in the list

bans

[in] The List of bans

param

[in] Pointer to user data. Passed through unmodified from the initiating function.

Remarks

The **chatEnumChannelBansCallback** function is called when an attempt to enumerate channel bans has completed. The available results are whether the attempt was successful, the list of the bans, number of bans, the channel that the attempt was made on.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_U
chatEnumChannelBansCallback	chatEnumChannelBansCallbackA	chatE

chatEnumChannelBansCallbackW and **chatEnumChannelBansCallbackA** are UNICODE and ANSI mapped versions of **chatEnumChannelBansCallback**. The arguments of **chatEnumChannelBansCallbackA** are ANSI strings; those of **chatEnumChannelBansCallbackW** are wide-character strings.

Section Reference: [Gamespy Chat SDK](#)

chatEnumChannelsCallbackAll

Called when an attempt to enumerate all the channels is complete.

```
typedef void (*chatEnumChannelsCallbackAll)(  
    CHAT chat,  
    CHATBool success,  
    int numChannels,  
    const gsi_char ** channels,  
    const gsi_char ** topics,  
    int * numUsers,  
    void * param );
```

Routine	Required Header	Distribution
chatEnumChannelsCallbackAll	<chat.h>	SDKZIP

Parameters

chat

[in] The initialized chat interface object.

success

[in] CHATTrue if success, CHATFalse if failure.

numChannels

[in] The number of channels in the list

channels

[in] The List of channels

topics

[in] The List of topics associated with the list of channels

numUsers

[in] The number of users for each channel

param

[in] Pointer to user data. Passed through unmodified from the initiating function.

Remarks

The **chatEnumChannelsCallbackAll** function is called when an enumeration attempt of all channels has completed. The function will contain all the data necessary to update the list of channels including names of channels, number of people in each channel, and channel topics. It is also called after each is enumerated (**chatEnumChannelsCallbackEach**).

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNI
chatEnumChannelsCallbackAll	chatEnumChannelsCallbackAllA	chatEnur

chatEnumChannelsCallbackAllW and **chatEnumChannelsCallbackAllA** are UNICODE and ANSI mapped versions of **chatEnumChannelsCallbackAll**. The arguments of **chatEnumChannelsCallbackAllA** are ANSI strings; those of **chatEnumChannelsCallbackAllW** are wide-character strings.

Section Reference: [Gamespy Chat SDK](#)

chatEnumChannelsCallbackEach

Called after an attempt to enumerate each channel.

```
typedef void (*chatEnumChannelsCallbackEach)(  
    CHAT chat,  
    CHATBool success,  
    int index,  
    const gsi_char * channel,  
    const gsi_char * topic,  
    int numUsers,  
    void * param );
```

Routine	Required Header	Distribution
chatEnumChannelsCallbackEach	<chat.h>	SDKZIP

Parameters

chat

[in] The initialized chat interface object.

success

[in] CHATTrue if success, CHATFalse if failure.

index

[in] The index of this channel

channel

[in] The name of the channel

topic

[in] A string containing the topic of the channel

numUsers

[in] The number of users in this channel

param

[in] Pointer to user data. Passed through unmodified from the initiating function.

Remarks

The **chatEnumChannelsCallbackEach** function is called when an attempt to enumerate each channel on the server is complete. The successful attempt will have a channel with an index, the name of the channel, the topic for that channel, the number of users.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI
chatEnumChannelsCallbackEach	chatEnumChannelsCallbackEachA	cha

chatEnumChannelsCallbackEachW and **chatEnumChannelsCallbackEachA** are UNICODE and ANSI mapped versions of **chatEnumChannelsCallbackEach**. The arguments of **chatEnumChannelsCallbackEachA** are ANSI strings; those of **chatEnumChannelsCallbackEachW** are wide-character strings.

Section Reference: [Gamespy Chat SDK](#)

chatEnumJoinedChannelsCallback

Called after an attempt to enumerate joined channels.

```
typedef void (*chatEnumJoinedChannelsCallback)(  
    CHAT chat,  
    int index,  
    const gsi_char * channel,  
    void * param );
```

Routine	Required Header	Distribution
chatEnumJoinedChannelsCallback	<chat.h>	SDKZIP

Parameters

chat

[in] The initialized chat interface object.

index

[in] An index of the joined channels for this channel

channel

[in] The name of the channel

param

[in] Pointer to user data. Passed through unmodified from the initiating function.

Remarks

The **chatEnumJoinedChannelsCallback** function is called when an attempt to enumerate--the channels the local player has joined--is complete. The function will contain the channel name, an index to the channel which refers to the position in the list of joined channels.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	(
chatEnumJoinedChannelsCallback	chatEnumJoinedChannelsCallbackA	c

chatEnumJoinedChannelsCallbackW and **chatEnumJoinedChannelsCallbackA** are UNICODE and ANSI mapped versions of **chatEnumJoinedChannelsCallback**. The arguments of **chatEnumJoinedChannelsCallbackA** are ANSI strings; those of **chatEnumJoinedChannelsCallbackW** are wide-character strings.

Section Reference: [Gamespy Chat SDK](#)

chatEnumUsersCallback

Called after an attempt to enumerate the users in a channel.

```
typedef void (*chatEnumUsersCallback)(  
    CHAT chat,  
    CHATBool success,  
    const gsi_char * channel,  
    int numUsers,  
    const gsi_char ** users,  
    int * modes,  
    void * param );
```

Routine	Required Header	Distribution
chatEnumUsersCallback	<chat.h>	SDKZIP

Parameters

chat

[in] The initialized chat interface object.

success

[in] CHATTrue if success, CHATFalse if failure.

channel

[in] The name of the channel

numUsers

[in] The number of users in the channel

users

[in] The list of users names in the channel

modes

[in] The list of modes for the channel

param

[in] Pointer to user data. Passed through unmodified from the initiating function.

Remarks

The **chatEnumUsersCallback** is called when an attempt to enumerate all of the users in a given channel is made. The function will have the information of the users in the channel if success is CHATTrue.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defir
chatEnumUsersCallback	chatEnumUsersCallbackA	chatEnumUsersCallb

chatEnumUsersCallbackW and **chatEnumUsersCallbackA** are UNICODE and ANSI mapped versions of **chatEnumUsersCallback**. The arguments of **chatEnumUsersCallbackA** are ANSI strings; those of **chatEnumUsersCallbackW** are wide-character strings.

Section Reference: [Gamespy Chat SDK](#)

chatFillInUserCallback

Used in conjunction with the chatConnectSpecial and chatConnectSecure functions; called to fill in the user field after the actual network connection to the chat server has been made.

```
typedef void (*chatFillInUserCallback)(  
    CHAT chat,  
    unsigned int IP,  
    gsi_char user[128],  
    void * param );
```

Routine	Required Header	Distribution
chatFillInUserCallback	<chat.h>	SDKZIP

Parameters

chat

[in] The initialized chat interface object.

IP

[in] The IP address in string form: "xxx.xxx.xxx.xxx" to encode

user

[in] The user name to encode

param

[in] Pointer to user data. The same param that was passed to chatConnectSecure or chatConnectSpecial.

Remarks

This is used by the Peer SDK to encode the local machine's IP address (as known to the chat server) in the user field.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
chatFillInUserCallback	chatFillInUserCallbackA	chatFillInUserCallbackW

chatFillInUserCallbackW and **chatFillInUserCallbackA** are UNICODE and ANSI mapped versions of **chatFillInUserCallback**. The arguments of **chatFillInUserCallbackA** are ANSI strings; those of **chatFillInUserCallbackW** are wide-character strings.

Section Reference: [Gamespy Chat SDK](#)

See Also: [chatConnectSecure](#), [chatConnectSpecial](#)

chatGetBasicUserInfoCallback

Called after an attempt to get basic information on a user.

```
typedef void (*chatGetBasicUserInfoCallback)(  
    CHAT chat,  
    CHATBool success,  
    const gsi_char *nick,  
    const gsi_char *user,  
    const gsi_char *address,  
    void *param );
```

Routine	Required Header	Distribution
chatGetBasicUserInfoCallback	<chat.h>	SDKZIP

Parameters

chat

[in] The initialized chat interface object.

success

[in] CHATTrue if success, CHATFalse if failure.

nick

[in] The user's chat nickname

user

[in] The nickname of the target user

address

[in] The IP address of the user

param

[in] Pointer to user data. Passed through unmodified from the initiating function.

Remarks

The **chatGetBasicUserInfoCallback** function is called when an attempt to get basic information on a user is completed. If successful, the information will contain the user's chat nickname, the IP address of that user.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE
chatGetBasicUserInfoCallback	chatGetBasicUserInfoCallbackA	chatGetBasicUserInfoCallbackW

chatGetBasicUserInfoCallbackW and **chatGetBasicUserInfoCallbackA** are UNICODE and ANSI mapped versions of **chatGetBasicUserInfoCallback**. The arguments of **chatGetBasicUserInfoCallbackA** are ANSI strings; those of **chatGetBasicUserInfoCallbackW** are wide-character strings.

Section Reference: [Gamespy Chat SDK](#)

chatGetChannelBasicUserInfoCallback

Called when an attempt to get everyone's basic user info is made.

```
typedef void (*chatGetChannelBasicUserInfoCallback)(  
    CHAT chat,  
    CHATBool success,  
    const gsi_char *channel,  
    const gsi_char *nick,  
    const gsi_char *user,  
    const gsi_char *address,  
    void *param );
```

Routine	Required Header	Distribution
chatGetChannelBasicUserInfoCallback	<chat.h>	SDKZIP

Parameters

chat

[in] The initialized chat interface object.

success

[in] CHATTrue if success, CHATFalse if failure.

channel

[in] The name of the channel

nick

[in] The local player's chat nickname

user

[in] The nickname of the target user

address

[in] The IP address of the target user

param

[in] Pointer to user data. Passed through unmodified from the initiating function.

Remarks

The **chatGetChannelBasicUserInfoCallback** function is called with a user's basic info for everyone in a channel. Called with a NULL nick/user/address at the end.

Unicode Mappings

Routine	GSI_UNICODE Not Defined
chatGetChannelBasicUserInfoCallback	chatGetChannelBasicUserInfoCa

chatGetChannelBasicUserInfoCallbackW and **chatGetChannelBasicUserInfoCallbackA** are UNICODE and ANSI mapped versions of **chatGetChannelBasicUserInfoCallback**. The arguments of **chatGetChannelBasicUserInfoCallbackA** are ANSI strings; those of **chatGetChannelBasicUserInfoCallbackW** are wide-character strings.

Section Reference: [Gamespy Chat SDK](#)

chatGetChannelKeysCallback

Called after an attempt to get the channel keys or user(s) keys.

```
typedef void (*chatGetChannelKeysCallback)(  
    CHAT chat,  
    CHATBool success,  
    const gsi_char * channel,  
    const gsi_char * user,  
    int num,  
    const gsi_char ** keys,  
    const gsi_char ** values,  
    void * param );
```

Routine	Required Header	Distribution
chatGetChannelKeysCallback	<chat.h>	SDKZIP

Parameters

chat

[in] The initialized chat interface object.

success

[in] CHATTrue if success, CHATFalse if failure.

channel

[in] The name of the channel

user

[in] The nickname of the target user

num

[in] The number of key/value pairs in the array

keys

[in] The array of key names whose values will be retrieved

values

[in] The array of values associated with the array of keys

param

[in] Pointer to user data. Passed through unmodified from the initiating function.

Remarks

The **chatGetChannelKeysCallback** function is called when an attempt to either get either the channel or user(s) keys is completed. If the call to chatGetChannelKeys was made on a set of users, then this function will get called for all users and have a NULL for "user" when done. If the call was for the channel keys, then the "user" will be NULL.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNIC
chatGetChannelKeysCallback	chatGetChannelKeysCallbackA	chatGetCh

chatGetChannelKeysCallbackW and **chatGetChannelKeysCallbackA** are UNICODE and ANSI mapped versions of **chatGetChannelKeysCallback**. The arguments of **chatGetChannelKeysCallbackA** are ANSI strings; those of **chatGetChannelKeysCallbackW** are wide-character strings.

Section Reference: [Gamespy Chat SDK](#)

chatGetChannelModeCallback

Called after an attempt to get the channel mode.

```
typedef void (*chatGetChannelModeCallback)(  
    CHAT chat,  
    CHATBool success,  
    const gsi_char *channel,  
    CHATChannelMode *mode,  
    void *param );
```

Routine	Required Header	Distribution
chatGetChannelModeCallback	<chat.h>	SDKZIP

Parameters

chat

[in] The initialized chat interface object.

success

[in] CHATTrue if success, CHATFalse if failure.

channel

[in] The name of channel

mode

[in] One of the predefined modes

param

[in] Pointer to user data. Passed through unmodified from the initiating function.

Remarks

The **chatGetChannelModeCallback** function is called when an attempt to get the channel mode is complete. If successful, the function will have the channel name and its mode.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNI
chatGetChannelModeCallback	chatGetChannelModeCallbackA	chatGetC

chatGetChannelModeCallbackW and **chatGetChannelModeCallbackA** are UNICODE and ANSI mapped versions of **chatGetChannelModeCallback**. The arguments of **chatGetChannelModeCallbackA** are ANSI strings; those of **chatGetChannelModeCallbackW** are wide-character strings.

Section Reference: [Gamespy Chat SDK](#)

chatGetChannelPasswordCallback

Called after an attempt to get the channel's password.

```
typedef void (*chatGetChannelPasswordCallback)(  
    CHAT chat,  
    CHATBool success,  
    const gsi_char *channel,  
    CHATBool enabled,  
    const gsi_char *password,  
    void *param );
```

Routine	Required Header	Distribution
chatGetChannelPasswordCallback	<chat.h>	SDKZIP

Parameters

chat

[in] The initialized chat interface object.

success

[in] CHATTrue if success, CHATFalse if failure.

channel

[in] The name of channel

enabled

[in] CHATTrue if enabled, CHATFalse if otherwise

password

[in] The channel password

param

[in] Pointer to user data. Passed through unmodified from the initiating function.

Remarks

The **chatGetChannelPasswordCallback** function is called when an attempt to obtain the channel's password is complete. If successful, the password for that channel will be available.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	C
chatGetChannelPasswordCallback	chatGetChannelPasswordCallbackA	c

chatGetChannelPasswordCallbackW and **chatGetChannelPasswordCallbackA** are UNICODE and ANSI mapped versions of **chatGetChannelPasswordCallback**. The arguments of **chatGetChannelPasswordCallbackA** are ANSI strings; those of **chatGetChannelPasswordCallbackW** are wide-character strings.

Section Reference: [Gamespy Chat SDK](#)

chatGetChannelTopicCallback

Called after an attempt to get the channel's topic.

```
typedef void (*chatGetChannelTopicCallback)(  
    CHAT chat,  
    CHATBool success,  
    const gsi_char *channel,  
    const gsi_char *topic,  
    void *param );
```

Routine	Required Header	Distribution
chatGetChannelTopicCallback	<chat.h>	SDKZIP

Parameters

chat

[in] The initialized chat interface object.

success

[in] CHATTrue if success, CHATFalse if failure.

channel

[in] The name of channel

topic

[in] A string containing the topic

param

[in] Pointer to user data. Passed through unmodified from the initiating function.

Remarks

The **chatGetChannelTopicCallback** function is called when an attempt to obtain the channel's topic is complete. If successful, the text message containing the topic for that channel will be available.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNIC
chatGetChannelTopicCallback	chatGetChannelTopicCallbackA	chatGetCl

chatGetChannelTopicCallbackW and **chatGetChannelTopicCallbackA** are UNICODE and ANSI mapped versions of **chatGetChannelTopicCallback**. The arguments of **chatGetChannelTopicCallbackA** are ANSI strings; those of **chatGetChannelTopicCallbackW** are wide-character strings.

Section Reference: [Gamespy Chat SDK](#)

chatGetGlobalKeysCallback

Called after an attempt to get the global keys for the user(s).

```
typedef void (*chatGetGlobalKeysCallback)(  
    CHAT chat,  
    CHATBool success,  
    const gsi_char *user,  
    int num,  
    const gsi_char **keys,  
    const gsi_char **values,  
    void *param );
```

Routine	Required Header	Distribution
chatGetGlobalKeysCallback	<chat.h>	SDKZIP

Parameters

chat

[in] The initialized chat interface object.

success

[in] CHATTrue if success, CHATFalse if failure.

user

[in] The nickname of the target user or the name of the channel

num

[in] The number of key/value pairs in the array

keys

[in] The array of key names whose values will be retrieved

values

[in] The array of values associated with the key array

param

[in] Pointer to user data. Passed through unmodified from the initiating function.

Remarks

The **chatGetGlobalKeysCallback** function is called when an attempt to obtain the global keys of a user or all users is complete. If successful, the keys for those user(s) will be available.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICOD
chatGetGlobalKeysCallback	chatGetGlobalKeysCallbackA	chatGetGlobal

chatGetGlobalKeysCallbackW and **chatGetGlobalKeysCallbackA** are UNICODE and ANSI mapped versions of **chatGetGlobalKeysCallback**. The arguments of **chatGetGlobalKeysCallbackA** are ANSI strings; those of **chatGetGlobalKeysCallbackW** are wide-character strings.

Section Reference: [Gamespy Chat SDK](#)

chatGetUserInfoCallback

Called after an attempt to get user information.

```
typedef void (*chatGetUserInfoCallback)(  
    CHAT chat,  
    CHATBool success,  
    const gsi_char *nick,  
    const gsi_char *user,  
    const gsi_char *name,  
    const gsi_char *address,  
    int numChannels,  
    const gsi_char **channels,  
    void *param );
```

Routine	Required Header	Distribution
chatGetUserInfoCallback	<chat.h>	SDKZIP

Parameters

chat

[in] The initialized chat interface object.

success

[in] CHATTrue if success, CHATFalse if failure.

nick

[in] The local player's chat nickname

user

[in] The nickname of the target user

name

[in] The name of the user to get info from

address

[in] The IP address of the user

numChannels

[in] The number of channels the user is in

channels

[in] The actual list of channels the user is in

param

[in] Pointer to user data. Passed through unmodified from the initiating function.

Remarks

The **chatGetUserInfoCallback** function is called when an attempt to get the user information about another player is completed. If successful, the user's nickname, IP address, the channels s/he is on will be available.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defi
chatGetUserInfoCallback	chatGetUserInfoCallbackA	chatGetUserInfoCall

chatGetUserInfoCallbackW and **chatGetUserInfoCallbackA** are UNICODE and ANSI mapped versions of **chatGetUserInfoCallback**. The arguments of **chatGetUserInfoCallbackA** are ANSI strings; those of **chatGetUserInfoCallbackW** are wide-character strings.

Section Reference: [Gamespy Chat SDK](#)

chatGetUserModeCallback

Called after an attempt to get the user's mode.

```
typedef void (*chatGetUserModeCallback)(  
    CHAT chat,  
    CHATBool success,  
    const gsi_char * channel,  
    const gsi_char * user,  
    int mode,  
    void * param );
```

Routine	Required Header	Distribution
chatGetUserModeCallback	<chat.h>	SDKZIP

Parameters

chat

[in] The initialized chat interface object.

success

[in] CHATTrue if success, CHATFalse if failure.

channel

[in] The name of channel

user

[in] The nickname of the target user

mode

[in] One of the predefined modes

param

[in] Pointer to user data. Passed through unmodified from the initiating function.

Remarks

The **chatGetUserModeCallback** function is called when an attempt to get the user mode is completed. If successful, the user's nickname and mode will be available.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE I
chatGetUserModeCallback	chatGetUserModeCallbackA	chatGetUserMoc

chatGetUserModeCallbackW and **chatGetUserModeCallbackA** are UNICODE and ANSI mapped versions of **chatGetUserModeCallback**. The arguments of **chatGetUserModeCallbackA** are ANSI strings; those of **chatGetUserModeCallbackW** are wide-character strings.

Section Reference: [Gamespy Chat SDK](#)

chatInvited

Used in conjunction with the chatConnect functions; called when the local user gets invited to a channel.

```
typedef void (*chatInvited)(  
    CHAT chat,  
    const gsi_char * channel,  
    const gsi_char * user,  
    void * param );
```

Routine	Required Header	Distribution
chatInvited	<chat.h>	SDKZIP

Parameters

chat

[in] The initialized chat interface object.

channel

[in] The channel to which this user was invited.

user

[in] The user who offered the invite.

param

[in] Pointer to user data. The same param that was passed to chatConnect through the callback structure.

Section Reference: [Gamespy Chat SDK](#)

See Also: [chatConnect](#), [chatInviteUser](#)

chatKicked

Used in conjunction with chatEnterChannel; called when the local user gets kicked from the channel.

```
typedef void (*chatKicked)(  
    CHAT chat,  
    const gsi_char * channel,  
    const gsi_char * user,  
    const gsi_char * reason,  
    void * param );
```

Routine	Required Header	Distribution
chatKicked	<chat.h>	SDKZIP

Parameters

chat

[in] The initialized chat interface object.

channel

[in] The channel the local player is in.

user

[in] The nickname of the user being kicked from the channel.

reason

[in] The same reason string sent into chatKickUser.

param

[in] Pointer to user data. The same param that was passed to chatEnterChannel through the callbacks structure.

Section Reference: [Gamespy Chat SDK](#)

See Also: [chatEnterChannel](#), [chatKickUser](#)

chatNewUserList

Used in conjunction with chatEnterChannel; Called when the chat server sends an entire new user list for a channel we're in.

```
typedef void (*chatNewUserList)(  
    CHAT chat,  
    const gsi_char *channel,  
    int num,  
    const gsi_char **users,  
    int *modes,  
    void *param );
```

Routine	Required Header	Distribution
chatNewUserList	<chat.h>	SDKZIP

Parameters

chat

[in] The initialized chat interface object.

channel

[in] The channel the local player is in.

num

[in] The number of users in the list.

users

[in] List of users.

modes

[in] List of user modes.

param

[in] Pointer to user data. The same param that was passed to chatEnterChannel through the callbacks structure.

Section Reference: [Gamespy Chat SDK](#)

chatNickErrorCallback

Used in conjunction with the chatConnect functions; called if there was an error with the provided nickname.

```
typedef void (*chatNickErrorCallback)(  
    CHAT chat,  
    int type,  
    const gsi_char * nick,  
    int numSuggestedNicks,  
    const gsi_char ** suggestedNicks,  
    void * param );
```

Routine	Required Header	Distribution
chatNickErrorCallback	<chat.h>	SDKZIP

Parameters

chat

[in] The initialized chat interface object.

type

[in] The type of error: indicates whether the nick was invalid or if it was in use.

nick

[in] The problematic nickname.

numSuggestedNicks

[in] The number of suggested nicknames.

suggestedNicks

[in] A list of suggested alternative nicknames.

param

[in] Pointer to user data. The same param that was passed to chatConnect.

Remarks

Suggested nicks are only provided if type is
CHAT_INVALID_UNIQUNICK.

Use chatRetryWithNick to continue the connect attempt with a new
nickname; otherwise, call chatDisconnect to stop the connection.

Section Reference: [Gamespy Chat SDK](#)

See Also: [chatConnect](#), [chatRetryWithNick](#), [chatRegisterUniqueNick](#),
[chatDisconnect](#)

chatPrivateMessage

Used in conjunction with the chatConnect functions; called when a message is received from another user.

```
typedef void (*chatPrivateMessage)(  
    CHAT chat,  
    const gsi_char *user,  
    const gsi_char *message,  
    int type,  
    void *param );
```

Routine	Required Header	Distribution
chatPrivateMessage	<chat.h>	SDKZIP

Parameters

chat

[in] The initialized chat interface object.

user

[in] The user who sent the message.

message

[in] The text of the message.

type

[in] The type of message.

param

[in] Pointer to user data. The same param that was passed to chatConnect through the callback structure.

Remarks

If user is NULL, this is a message from the server.

Section Reference: [Gamespy Chat SDK](#)

See Also: [chatConnect](#)

chatRaw

Used in conjunction with the chatConnect functions; all raw incoming network traffic gets passed to this function.

```
typedef void (*chatRaw)(  
    CHAT chat,  
    const gsi_char * raw,  
    void * param );
```

Routine	Required Header	Distribution
chatRaw	<chat.h>	SDKZIP

Parameters

chat

[in] The initialized chat interface object.

raw

[in] The raw data.

param

[in] Pointer to user data. The same param that was passed to chatConnect through the callbacks structure.

Section Reference: [Gamespy Chat SDK](#)

See Also: [chatConnect](#), [chatSendRaw](#)

chatTopicChanged

Used in conjunction with chatEnterChannel; called when the channel topic changes.

```
typedef void (*chatTopicChanged)(  
    CHAT chat,  
    const gsi_char * channel,  
    const gsi_char * topic,  
    void * param );
```

Routine	Required Header	Distribution
chatTopicChanged	<chat.h>	SDKZIP

Parameters

chat

[in] The initialized chat interface object.

channel

[in] The channel the local player is in.

topic

[in] The new topic (description) of the channel.

param

[in] Pointer to user data. The same param that was passed to chatEnterChannel through the callbacks structure.

Section Reference: [Gamespy Chat SDK](#)

See Also: [chatEnterChannel](#), [chatSetChannelTopic](#)

chatUserChangedNick

Used in conjunction with chatEnterChannel; called when a user in the channel changes their nickname.

```
typedef void (*chatUserChangedNick)(  
    CHAT chat,  
    const gsi_char * channel,  
    const gsi_char * oldNick,  
    const gsi_char * newNick,  
    void * param );
```

Routine	Required Header	Distribution
chatUserChangedNick	<chat.h>	SDKZIP

Parameters

chat

[in] The initialized chat interface object.

channel

[in] The channel the local player is in.

oldNick

[in] The old nickname of the user.

newNick

[in] The new nickname.

param

[in] Pointer to user data. The same param that was passed to chatEnterChannel through the callbacks structure.

Section Reference: [Gamespy Chat SDK](#)

See Also: [chatEnterChannel](#), [chatChangeNick](#)

chatUserJoined

Used in conjunction with chatEnterChannel; called when a user joins the channel.

```
typedef void (*chatUserJoined)(  
    CHAT chat,  
    const gsi_char *channel,  
    const gsi_char *user,  
    int mode,  
    void *param );
```

Routine	Required Header	Distribution
chatUserJoined	<chat.h>	SDKZIP

Parameters

chat

[in] The initialized chat interface object.

channel

[in] The channel the local player is in.

user

[in] The nickname of the joining user.

mode

[in] The joining user's mode.

param

[in] Pointer to user data. The same param that was passed to chatEnterChannel through the callbacks structure.

Section Reference: [Gamespy Chat SDK](#)

See Also: [chatEnterChannel](#)

chatUserListUpdated

Used in conjunction with chatEnterChannel; called when the channel's user list changes.

```
typedef void (*chatUserListUpdated)(  
    CHAT chat,  
    const gsi_char * channel,  
    void * param );
```

Routine	Required Header	Distribution
chatUserListUpdated	<chat.h>	SDKZIP

Parameters

chat

[in] The initialized chat interface object.

channel

[in] The channel the local player is in.

param

[in] Pointer to user data. The same param that was passed to chatEnterChannel through the callbacks structure.

Section Reference: [Gamespy Chat SDK](#)

See Also: [chatEnterChannel](#)

chatUserModeChanged

Used in conjunction with chatEnterChannel; called when the mode of a user in the channel changes.

```
typedef void (*chatUserModeChanged)(  
    CHAT chat,  
    const gsi_char *channel,  
    const gsi_char *user,  
    int mode,  
    void *param );
```

Routine	Required Header	Distribution
chatUserModeChanged	<chat.h>	SDKZIP

Parameters

chat

[in] The initialized chat interface object.

channel

[in] The channel the local player is in.

user

[in] The nickname of the user whose mode changed.

mode

[in] The new mode of the user.

param

[in] Pointer to user data. The same param that was passed to chatEnterChannel through the callbacks structure.

Section Reference: [Gamespy Chat SDK](#)

See Also: [chatEnterChannel](#), [chatSetUserMode](#)

chatUserParted

Used in conjunction with chatEnterChannel; called when a user parts the channel.

```
typedef void (*chatUserParted)(  
    CHAT chat,  
    const gsi_char *channel,  
    const gsi_char *user,  
    int why,  
    const gsi_char *reason,  
    const gsi_char *kicker,  
    void *param );
```

Routine	Required Header	Distribution
chatUserParted	<chat.h>	SDKZIP

Parameters

chat

[in] The initialized chat interface object.

channel

[in] The channel the local player is in.

user

[in] The nickname of the parting user.

why

[in] Code indicating reason user parted.

reason

[in] Explanation string.

kicker

[in] If reason is "kicked", identifies the kicker.

param

[in] Pointer to user data. The same param that was passed to chatEnterChannel through the callbacks structure.

Section Reference: [Gamespy Chat SDK](#)

See Also: [chatEnterChannel](#), [chatKickUser](#)

Chat SDK Structures

chatChannelCallbacks	A channel's callbacks.
CHATChannelMode	The mode settings of a chat channel.
chatGlobalCallbacks	A connection's global callbacks.

chatChannelCallbacks

A channel's callbacks.

```
typedef struct  
{  
    chatChannelMessage channelMessage;  
    chatKicked kicked;  
    chatUserJoined userJoined;  
    chatUserParted userParted;  
    chatUserChangedNick userChangedNick;  
    chatTopicChanged topicChanged;  
    chatChannelModeChanged channelModeChanged;  
    chatUserModeChanged userModeChanged;  
    chatUserListUpdated userListUpdated;  
    chatNewUserList newUserList;  
    chatBroadcastKeyChanged broadcastKeyChanged;  
    void * param;  
} chatChannelCallbacks;
```

Members

channelMessage

Called when a message is received in a channel.

kicked

Called when the local user is kicked from a channel.

userJoined

Called when a user joins a channel we're in.

userParted

Called when a user parts a channel we're in.

userChangedNick

Called when a user in a channel we're in changes nicks.

topicChanged

Called when the topic changes in a channel we're in.

channelModeChanged

Called when the mode changes in a channel we're in.

userModeChanged

Called when a user's mode changes in a channel we're in.

userListUpdated

Called when the user list changes (due to a join or a part) in a channel we're in.

newUserList

Called when the chat server sends an entire new user list for a channel we're in.

broadcastKeyChanged

Called when a user changes a broadcast key in a channel we're in.

param

A pointer to data that will be passed into each of the callbacks when triggered.

Section Reference: [Gamespy Chat SDK](#)

CHATChannelMode

The mode settings of a chat channel.

```
typedef struct  
{  
    CHATBool InviteOnly;  
    CHATBool Private;  
    CHATBool Secret;  
    CHATBool Moderated;  
    CHATBool NoExternalMessages;  
    CHATBool OnlyOpsChangeTopic;  
    int Limit;  
} CHATChannelMode;
```

Members

InviteOnly

Channel is invite-only.

Private

Channel is private.

Secret

Channel is secret.

Moderated

Channel is moderated,.

NoExternalMessages

External messages to channel are not allowed.

OnlyOpsChangeTopic

Topic is limited; only chanops may change it.

Limit

The maximum number of of users allowed.

Section Reference: [Gamespy Chat SDK](#)

See Also: [chatGetChannelMode](#), [chatSetChannelMode](#)

chatGlobalCallbacks

A connection's global callbacks.

```
typedef struct  
{  
    chatRaw raw;  
    chatDisconnected disconnected;  
    chatPrivateMessage privateMessage;  
    chatInvited invited;  
    void * param;  
} chatGlobalCallbacks;
```

Members

raw

Gets raw incoming network traffic.

disconnected

Called when the user has been disconnected.

privateMessage

Called when a private message from another user is received.

invited

Called when invited into a channel.

param

A pointer to data that will be passed into each of the callbacks when triggered.

Section Reference: [Gamespy Chat SDK](#)

Chat SDK Enumerations

[CHATBool](#)

Standard Boolean.

[CHATEnterResult](#)

The result of a channel enter attempt, passed into the chatEnterChannelCallback.

CHATBool

Standard Boolean.

```
typedef enum  
{  
    CHATFalse,  
    CHATTrue  
} CHATBool;
```

Constants

CHATFalse
False.

CHATTrue
True.

Section Reference: [Gamespy Chat SDK](#)

CHATEnterResult

The result of a channel enter attempt, passed into the chatEnterChannelCallback.

typedef enum

```
{  
    CHATEnterSuccess,  
    CHATBadChannelName,  
    CHATChannellsFull,  
    CHATInviteOnlyChannel,  
    CHATBannedFromChannel,  
    CHATBadChannelPassword,  
    CHATTooManyChannels,  
    CHATEnterTimedOut,  
    CHATBadChannelMask  
} CHATEnterResult;
```

Constants

CHATEnterSuccess

The channel was successfully entered.

CHATBadChannelName

The channel name was invalid.

CHATChannellsFull

The channel is at its user limit.

CHATInviteOnlyChannel

The channel is invite only.

CHATBannedFromChannel

The local user is banned from this channel.

CHATBadChannelPassword

The channel has a password, and a bad password (or none) was given.

CHATTooManyChannels

The server won't allow this user in any more channels.

CHATEnterTimedOut

The attempt to enter timed out.

Section Reference: [Gamespy Chat SDK](#)

See Also: [chatEnterChannelCallback](#)

HTTP SDK

Overview

The GameSpy HTTP SDK (GHTTP) is a library for downloading files or other data from HTTP servers. Simply make a request, and the library will connect to the server and download the requested file using the HTTP 1.1 protocol. GHTTP also supports uploading (posting) files or other data to HTTP servers. The SDK is written in standard ANSI C and has been tested on Win32, Unix, Mac, and consoles. The library has been designed to be easy to use, fast, and memory efficient (particularly useful on console systems with tight memory requirements). Just include all of the source files in your project, and you can start easily downloading data from web servers.

The SDK also includes two samples. `ghttpc` is a simple ANSI C sample that makes some requests, then waits for them to complete. `ghttpmfc` is a Windows MFC sample that provides a GUI for experimenting with the SDK.

The rest of this document presents a simple, step-by-step set of instructions for using GHTTP. See the main `ghttp.h` header file for more detailed information on each function.

File Manifest

The following files should be included with this package. If any of the files are missing, please [contact devsupport@gamespy.com](mailto:devsupport@gamespy.com).

File	Description
<i>ghttp.h</i> prototyped here)	GameSpy HTTP header (all user functions are prototyped here)
<i>ghttpMain.c, h</i>	The main entry point for all GHTTP functions
<i>ghttpBuffer.c,h</i>	Code for buffering of incoming and outgoing data
<i>ghttpCallbacks.c,h</i>	Code for calling callbacks
<i>ghttpCommon.c,h</i>	Common utility code
<i>ghttpConnection.c,h</i>	This code manages all current connections
<i>ghttpPost.c,h</i> (uploads)	Code for managing and procesing posts
<i>ghttpProcess.c,h</i> and state)	Code for processing requests (based on type and state)
<i>nonport.c,h</i>	Platform-specific code
<i>/ghttpc/</i>	ANSI-C sample
<i>/ghttpmfc/</i>	Windows MFC sample

Implementation

Step 1: Startup

To initialize the GHTTP SDK, call `ghttpStartup`. This can be called multiple times. There must, however, be a matching `ghttpCleanup` for each call. Note: it will be called automatically if a request function is called first.

Step 2: Make A Request

Once the library has been started up, its ready to start making requests. This is done by passing a URL to a GHTTP function, which then contacts the appropriate HTTP server, makes a request, then possibly downloads a file (or other resource).

There are five types of request, each of which has a basic function and an extended function: `get`, `save`, `stream`, `head`, and `post`.

get

The "get" type of function simply downloads the file into memory. This memory can be provided by the application, or it can be allocated by the library.

save

The "save" type of function saves the file directly to disk. The filename to save it as is passed into the request function.

stream

The "stream" type of function doesn't store the file at all. It calls an application-provided callback whenever part of the file is received from the server, and the application can then do what it wants with the data.

head

The "head" type of function is used when an application wants the headers that would normally be returned as part of a "get" request, but without actually getting the file.

post

The "post" type of function is used solely to post data, ignoring any

possible body returned by the server (response status and headers can still be checked). The get, save, and stream types can also optionally post data.

get

```
GHTTPRequest ghttpGet
(
    const char * URL,
    GHTTPBool blocking,
    ghttpCompletedCallback completedCallback,
    void * param
);
GHTTPRequest ghttpGetEx
(
    const char * URL,
    const char * headers,
    char * buffer,
    int bufferSize,
    GHTTPPost post,
    GHTTPBool throttle,
    GHTTPBool blocking,
    ghttpProgressCallback progressCallback,
    ghttpCompletedCallback completedCallback,
    void * param
);
```

URL

This is the URL for the file (i.e., "http://host.domain[:port]/path/filename").

headers

If not **NULL**, this is a string containing extra headers to send with the request.

buffer

The buffer to download to. If **NULL**, one will be allocated by the library.

bufferSize

If buffer is not `NULL`, the size of the buffer. If buffer is `NULL`, this should be 0.

post

If not `NULL`, post this object along with the request.

throttle

If `GHTTPTrue`, throttle this request's download speed.

blocking

If `GHTTPTrue`, the request function won't return until the request has finished.

progressCallback

If not `NULL`, gets called whenever the download progresses.

completedCallback

If not `NULL`, gets called when the download is completed (successfully or not).

param

This is optional user-data that will be passed into the callbacks.

save

```
GHTTPRequest ghttpSave
(
    const char * URL,
    const char * filename,
    GHTTPBool blocking,
    ghttpCompletedCallback completedCallback,
    void * param
);
GHTTPRequest ghttpSaveEx
(
    const char * URL,
    const char * filename,
    const char * headers,
    GHTTPPost post,
    GHTTPBool throttle,
```

```
GHTTPOptions blocking,  
ghttpProgressCallback progressCallback,  
ghttpCompletedCallback completedCallback,  
void * param  
);
```

URL

This is the URL for the file (i.e., "http://host.domain[:port]/path/filename").

filename

The filename to save the file as. Cannot be `NULL`.

headers

If not `NULL`, this is a string containing extra headers to send with the request.

post

If not `NULL`, post this object along with the request.

throttle

If `GHTTPOptionsTrue`, throttle this request's download speed.

blocking

If `GHTTPOptionsTrue`, the request function won't return until the request has finished.

progressCallback

If not `NULL`, gets called whenever the download progresses.

completedCallback

If not `NULL`, gets called when the download is completed (successfully or not).

param

This is optional user-data that will be passed into the callbacks.

stream

```
GHTTPOptions ghttpStream  
(
```

```

        const char * URL,
        GHTTPBool blocking,
        ghttpProgressCallback progressCallback,
        ghttpCompletedCallback completedCallback,
        void * param
    );
GHTTPRequest ghttpStreamEx
(
    const char * URL,
    const char * headers,
    GHTTPPost post,
    GHTTPBool throttle,
    GHTTPBool blocking,
    ghttpProgressCallback progressCallback,
    ghttpCompletedCallback completedCallback,
    void * param
);

```

URL

This is the URL for the file (i.e., "http://host.domain[:port]/path/filename").

headers

If not **NULL**, this is a string containing extra headers to send with the request.

post

If not **NULL**, post this object along with the request.

throttle

If **GHTTPTrue**, throttle this request's download speed.

blocking

If **GHTTPTrue**, the request function won't return until the request has finished.

progressCallback

If not **NULL**, gets called whenever the download progresses.

completedCallback

If not **NULL**, gets called when the download is completed

(successfully or not).

param

This is optional user-data that will be passed into the callbacks.

head

```
GHTTPRequest ghttpHead
(
    const char * URL,
    GHTTPBool blocking,
    ghttpCompletedCallback completedCallback,
    void * param
);
GHTTPRequest ghttpHeadEx
(
    const char * URL,
    const char * headers,
    GHTTPBool throttle,
    GHTTPBool blocking,
    ghttpProgressCallback progressCallback,
    ghttpCompletedCallback completedCallback,
    void * param
);
```

URL

This is the URL for the file (i.e., "http://host.domain[:port]/path/filename").

headers

If not `NULL`, this is a string containing extra headers to send with the request.

throttle

If `GHTTPTrue`, throttle this request's download speed.

blocking

If `GHTTPTrue`, the request function won't return until the request has finished.

progressCallback

If not `NULL`, gets called whenever the download progresses.

completedCallback

If not `NULL`, gets called when the download is completed (successfully or not).

param

This is optional user-data that will be passed into the callbacks.

post

```
GHTTPRequest ghttpPost
(
    const char * URL,
    GHTTPPost post,
    GHTTPBool blocking,
    ghttpCompletedCallback completedCallback,
    void * param
);
GHTTPRequest ghttpPostEx
(
    const char * URL,
    const char * headers,
    GHTTPPost post,
    GHTTPBool throttle,
    GHTTPBool blocking,
    ghttpProgressCallback progressCallback,
    ghttpCompletedCallback completedCallback,
    void * param
);
```

URL

This is the URL for the file (i.e., "http://host.domain[:port]/path/filename").

headers

If not `NULL`, this is a string containing extra headers to send with the request.

post

The object to post with the request. Cannot be `NULL`.

throttle

If `GHTTPTtrue`, throttle this request's download speed.

blocking

If `GHTTPTtrue`, the request function won't return until the request has finished.

progressCallback

If not `NULL`, gets called whenever the download progresses.

completedCallback

If not `NULL`, gets called when the download is completed (successfully or not).

param

This is optional user-data that will be passed into the callbacks.

Step 2: Wait For Callbacks

Any application that uses GHTTP in non-blocking mode (sets the blocking parameter to `GHTTPTfalse`) needs to call `ghttpThink` to let the library do any necessary processing. This call will process any current requests and call any callbacks if necessary. It will typically be called in the application's main loop. While it can be called as little as a few times a second, it should be called closer to 10-20 times a second. If downloading larger files, it may be desirable to call it even more often, to ensure that incoming buffers are emptied to make room for more incoming data.

Threads note: Making GHTTP requests concurrently from multiple threads is currently only supported under Win32. When using GHTTP from multiple threads, instead of calling `ghttpThink`, use `ghttpRequestThink` for each individual request. This allows that request's callback to be called from within the same thread in which it was started.

There are two callback types used by GHTTP: the "progress" callback, and the "completed" callback. The progress callback, if provided, gets

called when the state of the request changes and when file data is received from the server.

```
typedef enum
{
    GHTTPSocketInit,
    GHTTPhostLookup,
    GHTTPLookupPending,
    GHTTPConnecting,
    GHTTPSendingRequest,
    GHTTPPosting,
    GHTTPWaiting,
    GHTTPReceivingStatus,
    GHTTPReceivingHeaders,
    GHTTPReceivingFile
} GHTTPState;

typedef void (* ghttpProgressCallback)
(
    GHTTPRequest request,
    GHTTPState state,
    const char * buffer,
    int bufferLen,
    int bytesReceived,
    int totalSize,
    void * param
);
```

request

This is the same request identifier returned by the request function.

state

The current state of the request.

buffer

For get requests, the file so far. For save and stream, the most recent data received.

Header data is not passed into this callback. This will only be the actual file.

If state != `GHTTPReceivingFile`, this will be `NULL`.

`bufferLen`

The length of the data in buffer (buffer is also NUL-terminated).

If buffer is `NULL`, this will be 0.

`bytesRecieved`

If `GHTTPTrue`, the request function won't return until the request has finished.

`totalSize`

If not `NULL`, gets called whenever the download progresses.

`param`

This is optional user-data that will be passed into the callbacks.

Note: the state usually moves forward by one state at a time (i.e., `GHTTPSocketInit` -> `GHTTPHostLookup`). However, it will move from `GHTTPReceivingHeaders` back to `GHTTPSocketInit` if the request has been redirected, it will skip `GHTTPPosting` if not posting data, and it will move from `GHTTPReceivingHeaders` back to `GHTTPReceivingStatus` if it gets a 100-Continue status (this typically only happens while posting).

The completed callback gets called when the request is completed:

```
typedef enum
{
    GHTTPSuccess,
    GHTTPOutOfMemory,
    GHTTPBufferOverflow,
    GHTTTParseURLFailed,
    GHTTPHostLookupFailed,
    GHTTPSocketFailed,
    GHTTPConnectFailed,
    GHTTPBadResponse,
    GHTTPRequestRejected,
    GHTTPUnauthorized,
    GHTTPForbidden,
```

```

        GHTTPEFileNotFound,
        GHTTPEServerError,
        GHTTPEFileWriteFailed,
        GHTTPEFileReadFailed
    ) GHTTPEResult;

typedef GHTTPEBool (* ghttpCompletedCallback)
(
    GHTTPERequest request,
    GHTTPEResult result,
    char * buffer,
    int bufferLen,
    void * param
);

```

request

This is the same request identifier returned by the request function.

result

The result of the request.

buffer

If a get request, this is the entire file in memory. Otherwise, **NULL**.

bufferLen

The length of the file (even if not a get request).

param

This is optional user-data that will be passed into the callbacks.

The return value can be ignored if this is not a get request. For a get request, return **GHTTPETrue** to have the buffer's memory freed. If **GHTTPEFalse** is returned, it is the responsibility of the application to free the memory.

Step 4: Cleanup

When the application is done using GHTTP, call **ghttpCleanup** to free any resources it is using. This call can also be used if GHTTP will not be used for a while, and the application wishes to free up resources. If it is

called while requests are pending, they will be cancelled, and the completed callback will not be called.

Posting

[GHTTPPost](#) objects are used to post (upload) data along with a request. They can be used to upload simple string data, and they can be used to upload files. This allows for a range of uses, from posting to web forums to uploading custom skins. [GHTTPPost](#) objects can be passed to [ghttpGetEx](#), [ghttpSaveEx](#), and [ghttpStreamEx](#) to upload data and then receive a response from the server, or they can be passed to [ghttpPost](#) and [ghttpPostEx](#) to just upload data without getting a response.

[ghttpNewPost](#) is used to create a new [GHTTPPost](#) object. To add data to it, use [ghttpPostAddString](#), [ghttpPostAddFileFromDisk](#), and [ghttpPostAddFileFromMemory](#). Once an object is setup, it can be used in a request. An application must not modify a [GHTTPPost](#) object that is in the process of being used in a request. By default, the object will be automatically freed after being used. However, the same object can be used in multiple requests by calling [ghttpPostSetAutoFree](#) and setting the [autoFree](#) parameter to [GHTTPFalse](#). When done using the object, free it with [ghttpFreePost](#) (or set [autoFree](#) back to [GHTTPTrue](#) before using it for the last time). [ghttpPostSetCallback](#) can be used to setup a callback to be called whenever data is uploaded, which allows an application to monitor the progress of the upload in terms of both bytes and objects uploaded.

If only strings are being uploaded as part of a request, then it will be done using the "application/x-www-form-urlencoded" content type. If files are also being uploaded (either from disk or memory), then the post will use the "multipart/form-data" content type.

Miscellaneous

If throttling is enabled for a request, the download speed will be limited. To customize the throttle speed, use [ghttpThrottleSettings](#). To change a requests throttle setting after it has been started, use [ghttpSetThrottle"](#).

There is a known bug with Windows CE that causes it to return the wrong address when looking up certain DNS names (specifically, those with CNAME records). An example of a host name that CE will not handle correctly is "www.cnn.com".

GHTTP can handle HTTP redirection. If the server sends a response with a 3xx status code, and the new location is given, GHTTP will then attempt to open the new URL. The current state of the request will go from [GHTTPReceivingHeaders](#) to [GHTTPSocketInit](#).

A current request can be cancelled by passing its [GHTTPRequest](#) identifier (returned from the request function) to [ghttpCancelRequest](#). The completed callback will not be called for this request.

The current state of a request can be obtained at any time with [ghttpGetState](#).

If the state of a request has passed [GHTTPReceivingStatus](#), then [ghttpGetResponseStatus](#) can be used to get both the status code and status string returned by the HTTP server.

If the request has passed the [GHTTPReceivingHeaders](#) state, then [ghttpGetHeaders](#) can be called to get the headers returned by the server.

[ghttpGetURL](#) can be called to get the URL being retrieved by the request. If the request has been redirected, the URL returned will be the new URL, not the one passed into the request function.

All requests can be forwarded to a web proxy by passing a proxy's address to [ghttpSetProxy](#).

UNICODE Support

The GameSpy SDKs support an optional UNICODE interface for widestring applications. To use this interface, first define the symbol "[GSI_UNICODE](#)". Then, use widestrings wherever ANSI strings were previously called for. When in doubt, please refer to the header files for specific function declarations.

Although the GameSpy SDK interfaces support UNICODE parameters, some items may be stripped of their extra UNICODE information. These items include: nickname, email address, and URL strings. You may pass in widestring values, but they will first be converted to their ANSI counterparts before transmission.

HTTP SDK Functions

ghttpCancelRequest	Cancel a HTTP request in progress.
ghttpCleanup	Destruct the HTTP sdk. Free internally allocated memory.
ghttpFreePost	Free a post object.
ghttpGet	Make a HTTP GET request and save the response to memory.
ghttpGetEx	Make a HTTP GET request and save the response to memory.
ghttpGetHeaders	Get the response headers from an HTTP request.
ghttpGetResponseStatus	Get the response's status string and status code.
ghttpGetState	Obtain the current state of a request.
ghttpGetURL	Used to obtain the URL associated with a request.
ghttpHead	Make a HTTP HEAD request, which will only retrieve the response headers and not the normal response body.

ghttpHeadEx	Make a HTTP HEAD request, which will only retrieve the response headers and not the normal response body.
ghttpNewPost	Creates a new post object, which is used to represent data to send a web server as part of a request.
ghttpPost	Do a HTTP POST, which can be used to upload data to a web server.
ghttpPostAddFileFromDisk	Adds a disk file to the post object.
ghttpPostAddFileFromMemory	Adds a file, in memory, to the post object.
ghttpPostAddString	Adds a string to the post object.
ghttpPostEx	Do a HTTP POST, which can be used to upload data to a web server.
ghttpPostSetAutoFree	Sets a post object's auto-free flag.
ghttpPostSetCallback	Sets the callback for a post object.
ghttpRequestThink	Process just one particular request.
ghttpSave	Make a HTTP GET request and save the response to disk.

ghttpSaveEx	Make a HTTP GET request and save the response to disk.
ghttpSetMaxRecvTime	Used to throttle based on time, not on bandwidth.
ghttpSetProxy	Sets a proxy server address.
ghttpSetRequestProxy	Sets a proxy server for a specific request.
ghttpSetThrottle	Used to start/stop throttling an existing connection.
ghttpStartup	Initialize the HTTP SDK.
ghttpStream	Make a HTTP GET request and stream in the response without saving it in memory.
ghttpStreamEx	Make a HTTP GET request and stream in the response without saving it in memory.
ghttpThink	Processes all current http requests.
ghttpThrottleSettings	Used to adjust the throttle settings.

ghttpCancelRequest

Cancel a HTTP request in progress.

```
void ghttpCancelRequest(  
    GHTTPRequest request );
```

Routine	Required Header	Distribution
ghttpCancelRequest	<ghttp.h>	SDKZIP

Parameters

request

[in] A valid GHTTPRequest object.

Remarks

The GHTTPRequest should not be referenced once this function returns.

Section Reference: [Gamespy HTTP SDK](#)

ghttpCleanup

Destruct the HTTP sdk. Free internally allocated memory.

void ghttpCleanup();

Routine	Required Header	Distribution
ghttpCleanup	<ghttp.h>	SDKZIP

Remarks

One call to **ghttpCleanup** should be made for each call to ghttpStartup.

Section Reference: [Gamespy HTTP SDK](#)

See Also: [ghttpStartup](#)

ghttpFreePost

Free a post object.

```
void ghttpFreePost(  
    GHTTPPost post );
```

Routine	Required Header	Distribution
ghttpFreePost	<ghttp.h>	SDKZIP

Parameters

post

[in] Post object created with ghttpNewPost.

Remarks

By default, post objects created with `ghttpNewPost` will be automatically freed after being used in a request. However `ghttpPostSetAutoFree` can be used to turn off the post object's auto-free property. This can be useful if a single post object will be used in multiple requests. You should then use this function to manually free the post object after the last request it has been used in completes.

Section Reference: [Gamespy HTTP SDK](#)

See Also: [ghttpNewPost](#), [ghttpPostSetAutoFree](#)

ghttpGet

Make a HTTP GET request and save the response to memory.

```
GHTTPRequest ghttpGet(  
    const gsi_char * URL,  
    GHTTPBool blocking,  
    ghttpCompletedCallback completedCallback,  
    void * param );
```

Routine	Required Header	Distribution
ghttpGet	<ghttp.h>	SDKZIP

Return Value

If less than 0, the request failed and this is a `GHTTPRequestError` value. Otherwise it identifies the request.

Parameters

URL

[in] URL

blocking

[in] If true, this call doesn't return until the file has been received.

completedCallback

[in] Called when the file has been received.

param

[in] Optional free-format user data for use by the callback

Remarks

This function is used to download the contents of a web page to memory. The application can provide the memory by supplying a buffer to this function, or the SDK can be allocate the memory internally. Use **ghttpGetEx** for extra optionional parameters.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
ghttpGet	ghttpGetA	ghttpGetW

ghttpGetW and **ghttpGetA** are UNICODE and ANSI mapped versions of **ghttpGet**. The arguments of **ghttpGetA** are ANSI strings; those of **ghttpGetW** are wide-character strings.

Section Reference: [Gamespy HTTP SDK](#)

See Also: [ghttpGetEx](#), [ghttpSave](#), [ghttpStream](#), [ghttpHead](#), [ghttpPost](#)

ghttpGetEx

Make a HTTP GET request and save the response to memory.

```
GHTTPRequest ghttpGetEx(  
    const gsi_char * URL,  
    const gsi_char * headers,  
    char * buffer,  
    int bufferSize,  
    GHTTPPost post,  
    GHTTPBool throttle,  
    GHTTPBool blocking,  
    ghttpProgressCallback progressCallback,  
    ghttpCompletedCallback completedCallback,  
    void * param );
```

Routine	Required Header	Distribution
ghttpGetEx	<ghttp.h>	SDKZIP

Return Value

If less than 0, the request failed and this is a `GHTTPRequestError` value. Otherwise it identifies the request.

Parameters

URL

[in] URL

headers

[in] Optional headers to pass with the request. Can be NULL or "".

buffer

[in] Optional user-supplied buffer. Set to NULL to have one allocated. Must be (size+1) to allow null terminating character.

bufferSize

[in] The size of the user-supplied buffer in bytes. 0 if buffer is NULL.

post

[in] Optional data to be posted. Can be NULL.

throttle

[in] If true, throttle this connection's download speed.

blocking

[in] If true, this call doesn't return until the file has been received.

progressCallback

[in] Called periodically with progress updates. Can be NULL.

completedCallback

[in] Called when the file has been received. Can be NULL.

param

[in] Optional free-format user data for use by the callback

Remarks

This function is used to download the contents of a web page to memory. The application can provide the memory by supplying a buffer to this function, or the SDK can be allocate the memory internally. Use `ghttpGet` for a simpler version of this function.

Section Reference: [Gamespy HTTP SDK](#)

See Also: [ghttpGet](#), [ghttpSaveEx](#), [ghttpStreamEx](#), [ghttpHeadEx](#), [ghttpPostEx](#)

ghttpGetHeaders

Get the response headers from an HTTP request.

```
const char * ghttpGetHeaders(  
    GHTTPRequest request );
```

Routine	Required Header	Distribution
ghttpGetHeaders	<ghttp.h>	SDKZIP

Return Value

The headers returned in the response.

Parameters

request

[in] A valid request object

Remarks

Only valid if the request's state is GHTTPReceivingHeaders.

Section Reference: [Gamespy HTTP SDK](#)

See Also: [ghttpGetState](#)

ghttpGetResponseStatus

Get the response's status string and status code.

```
const char * ghttpGetResponseStatus(  
    GHTTPRequest request,  
    int * statusCode );
```

Routine	Required Header	Distribution
ghttpGetResponseStatus	<ghttp.h>	SDKZIP

Return Value

The response's status string.

Parameters

request

[in] A valid request object

statusCode

[out] Status code.

Remarks

Can only be used if the state has passed GHTTPReceivingStatus.
The status string is a user-readable representation of the result of the request.

The status code is a 3 digit number which can be used to get more details on the result of the request. There are 5 possible values for the first digit:

1xx: Informational

2xx: Success

3xx: Redirection

4xx: Client Error

5xx: Server Error

See RFC2616 (HTTP 1.1) and any follow-up RFCs for more information on specific codes.

Section Reference: [Gamespy HTTP SDK](#)

See Also: [ghttpGetState](#)

ghttpGetState

Obtain the current state of a request.

```
GHTTPState ghttpGetState(  
    GHTTPRequest request );
```

Routine	Required Header	Distribution
<code>ghttpGetState</code>	<code><ghttp.h></code>	SDKZIP

Return Value

The state of an HTTP request.

Parameters

request

[in] A valid request object

Section Reference: [Gamespy HTTP SDK](#)

ghttpGetURL

Used to obtain the URL associated with a request.

```
const char * ghttpGetURL(  
    GHTTPRequest request );
```

Routine	Required Header	Distribution
ghttpGetURL	<ghttp.h>	SDKZIP

Return Value

The URL associated with the request.

Parameters

request

[in] A valid request object

Remarks

If the request has been redirected, this function will return the new URL, not the original URL.

Section Reference: [Gamespy HTTP SDK](#)

ghttpHead

Make a HTTP HEAD request, which will only retrieve the response headers and not the normal response body.

```
GHTTPRequest ghttpHead(  
    const gsi_char * URL,  
    GHTTPBool blocking,  
    ghttpCompletedCallback completedCallback,  
    void * param );
```

Routine	Required Header	Distribution
ghttpHead	<ghttp.h>	SDKZIP

Return Value

If less than 0, the request failed and this is a `GHTTPRequestError` value. Otherwise it identifies the request.

Parameters

URL

[in] URL

blocking

[in] If true, this call doesn't return until finished

completedCallback

[in] Called when the request has finished.

param

[in] Optional free-format user data for use by the callback

Remarks

This function is similar to `ghttpGet`, except it only gets the response headers. This is done by making an HEAD request instead of a GET request, which instructs the HTTP server to leave the body out of the response.

Use `ghttpHeadEx` for extra optional parameters.

Section Reference: [Gamespy HTTP SDK](#)

See Also: [ghttpGet](#), [ghttpSave](#), [ghttpStream](#), [ghttpHeadEx](#), [ghttpPost](#)

ghttpHeadEx

Make a HTTP HEAD request, which will only retrieve the response headers and not the normal response body.

```
GHTTPRequest ghttpHeadEx(  
    const gsi_char * URL,  
    const gsi_char * headers,  
    GHTTPBool throttle,  
    GHTTPBool blocking,  
    ghttpProgressCallback progressCallback,  
    ghttpCompletedCallback completedCallback,  
    void * param );
```

Routine	Required Header	Distribution
ghttpHeadEx	<ghttp.h>	SDKZIP

Return Value

If less than 0, the request failed and this is a `GHTTPRequestError` value. Otherwise it identifies the request.

Parameters

URL

[in] URL

headers

[in] Optional headers to pass with the request. Can be NULL or "".

throttle

[in] If true, throttle this connection's download speed.

blocking

[in] If true, this call doesn't return until finished

progressCallback

[in] Called whenever new data is received. Can be NULL.

completedCallback

[in] Called when the request has finished. Can be NULL.

param

[in] Optional free-format user data for use by the callback

Remarks

This function is similar to `ghttpGetEx`, except it only gets the response headers. This is done by making an HEAD request instead of a GET request, which instructs the HTTP server to leave the body out of the response.

Use `ghttpHead` for a simpler version of this function.

Section Reference: [Gamespy HTTP SDK](#)

See Also: [ghttpGetEx](#), [ghttpSaveEx](#), [ghttpStreamEx](#), [ghttpHead](#), [ghttpPostEx](#)

ghttpNewPost

Creates a new post object, which is used to represent data to send a web server as part of a request.

GHTTPPost ghttpNewPost();

Routine	Required Header	Distribution
ghttpNewPost	<ghttp.h>	SDKZIP

Return Value

The newly created post object, or NULL if it cannot be created.

Remarks

After getting the post object, use the `ghttpPostAdd*()` functions to add data to the object, and `ghttpPostSetCallback()` to add a callback to monitor the progress of the data upload.

By default post objects automatically free themselves after posting. To use the same post with more than one request, set `auto-free` to `false`, then use `ghttpFreePost` to free it after every request its being used in is completed.

Section Reference: [Gamespy HTTP SDK](#)

See Also: [ghttpPostAddString](#), [ghttpPostAddFileFromDisk](#), [ghttpPostAddFileFromMemory](#), [ghttpPostSetAutoFree](#), [ghttpFreePost](#), [ghttpPostSetCallback](#)

ghttpPost

Do a HTTP POST, which can be used to upload data to a web server.

```
GHTTPRequest ghttpPost(  
    const gsi_char * URL,  
    GHTTPPost post,  
    GHTTPBool blocking,  
    ghttpCompletedCallback completedCallback,  
    void * param );
```

Routine	Required Header	Distribution
ghttpPost	<ghttp.h>	SDKZIP

Return Value

If less than 0, the request failed and this is a `GHTTPRequestError` value. Otherwise it identifies the request.

Parameters

URL

[in] URL

post

[in] The data to be posted.

blocking

[in] If true, this call doesn't return until finished

completedCallback

[in] Called when the file has finished streaming. Can be NULL.

param

[in] Optional free-format user data for use by the callback

Remarks

This function is used to post data to a web page, ignoring any possible response body sent by the server (response status and response headers can still be checked). If you want to post data and receive a response, use `ghttpGetEx`, `ghttpSaveEx`, or `ghttpStreamEx`. Use **`ghttpPostEx`** for extra optional parameters.

Section Reference: [Gamespy HTTP SDK](#)

See Also: [ghttpGet](#), [ghttpGetEx](#), [ghttpSave](#), [ghttpSaveEx](#), [ghttpStream](#), [ghttpStreamEx](#), [ghttpHead](#), [ghttpPostEx](#)

ghttpPostAddFileFromDisk

Adds a disk file to the post object.

```
GHTTPBool ghttpPostAddFileFromDisk(  
    GHTTPPost post,  
    const gsi_char * name,  
    const gsi_char * filename,  
    const gsi_char * reportFilename,  
    const gsi_char * contentType );
```

Routine	Required Header	Distribution
ghttpPostAddFileFromDisk	<ghttp.h>	SDKZIP

Return Value

GHTTPTrue if the file was added successfully.

Parameters

post

[in] Post object

name

[in] The name to attach to this file.

filename

[in] The name (and possibly path) to the file to upload.

reportFilename

[in] The filename given to the web server.

contentType

[in] The MIME type for this file.

Remarks

The reportFilename is what is reported to the server as the filename. If NULL or empty, the filename will be used (including any possible path).

The contentType is the MIME type to report for this file. If NULL, "application/octet-stream" is used.

The file isn't read from until the data is actually sent to the server.

When uploading files the content type of the overall request (as opposed to the content this of this file) will be "multipart/form-data".

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE I
ghhttpPostAddFileFromDisk	ghhttpPostAddFileFromDiskA	ghhttpPostAddFile

ghhttpPostAddFileFromDiskW and **ghhttpPostAddFileFromDiskA** are UNICODE and ANSI mapped versions of **ghhttpPostAddFileFromDisk**. The arguments of **ghhttpPostAddFileFromDiskA** are ANSI strings; those of **ghhttpPostAddFileFromDiskW** are wide-character strings.

Section Reference: [Gamespy HTTP SDK](#)

See Also: [ghhttpNewPost](#), [ghhttpPost](#), [ghhttpPostAddString](#), [ghhttpPostAddFileFromMemory](#)

ghttpPostAddFileFromMemory

Adds a file, in memory, to the post object.

```
GHTTPBool ghttpPostAddFileFromMemory(  
    GHTTPPost post,  
    const gsi_char * name,  
    const char * buffer,  
    int bufferLen,  
    const gsi_char * reportFilename,  
    const gsi_char * contentType );
```

Routine	Required Header	Distribution
ghttpPostAddFileFromMemory	<ghttp.h>	SDKZIP

Return Value

GHTTPTrue if the file was added successfully.

Parameters

post

[in] Post object

name

[in] The name to attach to this file.

buffer

[in] The data to send.

bufferLen

[in] The number of bytes of data to send.

reportFilename

[in] The filename given to the web server.

contentType

[in] The MIME type for this file.

Remarks

The reportFilename is what is reported to the server as the filename. It cannot be NULL or empty.

The contentType is the MIME type to report for this file. If NULL, "application/octet-stream" is used.

The data is not copied off in this call. The data pointer is read from as the data is actually sent to the server. The pointer must remain valid during requests.

When uploading files the content type of the overall request (as opposed to the content this of this file) will be "multipart/form-data".

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE
<code>gHttpPostAddFileFromMemory</code>	<code>gHttpPostAddFileFromMemoryA</code>	<code>gHttpPostAddFileFromMemoryW</code>

`gHttpPostAddFileFromMemoryW` and `gHttpPostAddFileFromMemoryA` are UNICODE and ANSI mapped versions of `gHttpPostAddFileFromMemory`. The arguments of `gHttpPostAddFileFromMemoryA` are ANSI strings; those of `gHttpPostAddFileFromMemoryW` are wide-character strings.

Section Reference: [Gamespy HTTP SDK](#)

See Also: [gHttpNewPost](#), [gHttpPost](#), [gHttpPostAddFileFromDisk](#), [gHttpPostAddString](#)

ghttpPostAddString

Adds a string to the post object.

```
GHTTPBool ghttpPostAddString(  
    GHTTPPost post,  
    const gsi_char * name,  
    const gsi_char * string );
```

Routine	Required Header	Distribution
ghttpPostAddString	<ghttp.h>	SDKZIP

Return Value

GHTTPTrue if the string was added successfully.

Parameters

post

[in] Post object

name

[in] The name to attach to this string.

string

[in] The string to send.

Remarks

If a post object only contains string, the content type for the upload will be the "application/x-www-form-urlencoded". If any files are added, the content type for the upload will become "multipart/form-data".

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
ghhttpPostAddString	ghhttpPostAddStringA	ghhttpPostAddStringW

ghhttpPostAddStringW and **ghhttpPostAddStringA** are UNICODE and ANSI mapped versions of **ghhttpPostAddString**. The arguments of **ghhttpPostAddStringA** are ANSI strings; those of **ghhttpPostAddStringW** are wide-character strings.

Section Reference: [Gamespy HTTP SDK](#)

See Also: [ghhttpNewPost](#), [ghhttpPost](#), [ghhttpPostAddFileFromDisk](#), [ghhttpPostAddFileFromMemory](#)

ghhttpPostEx

Do a HTTP POST, which can be used to upload data to a web server.

```
GHTTPRequest ghhttpPostEx(  
    const gsi_char * URL,  
    const gsi_char * headers,  
    GHTTPPost post,  
    GHTTPBool throttle,  
    GHTTPBool blocking,  
    ghhttpProgressCallback progressCallback,  
    ghhttpCompletedCallback completedCallback,  
    void * param );
```

Routine	Required Header	Distribution
ghhttpPostEx	<ghhttp.h>	SDKZIP

Return Value

If less than 0, the request failed and this is a `GHTTPRequestError` value. Otherwise it identifies the request.

Parameters

URL

[in] URL

headers

[in] Optional headers to pass with the request. Can be NULL or "".

post

[in] The data to be posted.

throttle

[in] If true, throttle this connection's download speed.

blocking

[in] If true, this call doesn't return until finished.

progressCallback

[in] Called whenever new data is received. Can be NULL.

completedCallback

[in] Called when the file has finished streaming. Can be NULL.

param

[in] Optional free-format user data for use by the callback

Remarks

This function is used to post data to a web page, ignoring any possible response body sent by the server (response status and response headers can still be checked). If you want to post data and receive a response, use [ghttpGetEx](#), [ghttpSaveEx](#), or [ghttpStreamEx](#). Use [ghttpPost](#) for a simpler version of this function.

Section Reference: [Gamespy HTTP SDK](#)

See Also: [ghttpGetEx](#), [ghttpSaveEx](#), [ghttpStreamEx](#), [ghttpHeadEx](#), [ghttpPost](#)

ghttpPostSetAutoFree

Sets a post object's auto-free flag.

```
void ghttpPostSetAutoFree(  
    GHTTPPost post,  
    GHTTPBool autoFree );
```

Routine	Required Header	Distribution
ghttpPostSetAutoFree	<ghttp.h>	SDKZIP

Parameters

post

[in] Post object

autoFree

[in] True if object should be auto-freed

Remarks

By default post objects automatically free themselves after posting. To use the same post with more than one request, set auto-free to false, then use `ghttpFreePost` to free it after every request it's being used in is completed.

Section Reference: [Gamespy HTTP SDK](#)

See Also: [ghttpNewPost](#), [ghttpFreePost](#), [ghttpPost](#)

ghttpPostSetCallback

Sets the callback for a post object.

```
void ghttpPostSetCallback(  
    GHTTPPost post,  
    ghttpPostCallback callback,  
    void * param );
```

Routine	Required Header	Distribution
ghttpPostSetCallback	<ghttp.h>	SDKZIP

Parameters

post

[in] The post object to set the callback on.

callback

[in] The callback to call when using this post object.

param

[in] User data passed to the callback.

Section Reference: [Gamespy HTTP SDK](#)

See Also: [ghttpNewPost](#)

ghttpRequestThink

Process just one particular request.

```
GHTTPBool ghttpRequestThink(  
    GHTTPRequest request );
```

Routine	Required Header	Distribution
ghttpRequestThink	<ghttp.h>	SDKZIP

Return Value

GHTTPFalse if the request cannot be found.

Parameters

request

[in] A valid request object to process.

Remarks

This allows an HTTP request to be processed in a separate thread (only supported under Win32).

Section Reference: [Gamespy HTTP SDK](#)

See Also: [ghttpThink](#)

ghttpSave

Make a HTTP GET request and save the response to disk.

```
GHTTPRequest ghttpSave(  
    const gsi_char * URL,  
    const gsi_char * filename,  
    GHTTPBool blocking,  
    ghttpCompletedCallback completedCallback,  
    void * param );
```

Routine	Required Header	Distribution
ghttpSave	<ghttp.h>	SDKZIP

Return Value

If less than 0, the request failed and this is a `GHTTPRequestError` value. Otherwise it identifies the request.

Parameters

URL

[in] URL

filename

[in] The path and name to store the file as locally.

blocking

[in] If true, this call doesn't return until the file has been received.

completedCallback

[in] Called when the file has been received. Can be NULL.

param

[in] Optional free-format user data for use by the callback

Remarks

This function is used to download the contents of a web page directly to disk. The application supplies the path and filename at which to save the response.

Use **ghttpSaveEx** for extra optional parameters.

Section Reference: [Gamespy HTTP SDK](#)

See Also: [ghttpGet](#), [ghttpSaveEx](#), [ghttpStream](#), [ghttpHead](#), [ghttpPost](#)

ghttpSaveEx

Make a HTTP GET request and save the response to disk.

```
GHTTPRequest ghttpSaveEx(  
    const gsi_char * URL,  
    const gsi_char * filename,  
    const gsi_char * headers,  
    GHTTPPost post,  
    GHTTPBool throttle,  
    GHTTPBool blocking,  
    ghttpProgressCallback progressCallback,  
    ghttpCompletedCallback completedCallback,  
    void * param );
```

Routine	Required Header	Distribution
ghttpSaveEx	<ghttp.h>	SDKZIP

Return Value

If less than 0, the request failed and this is a `GHTTPRequestError` value. Otherwise it identifies the request.

Parameters

URL

[in] URL

filename

[in] The path and name to store the file as locally.

headers

[in] Optional headers to pass with the request. Can be NULL or "".

post

[in] Optional data to be posted. Can be NULL.

throttle

[in] If true, throttle this connection's download speed.

blocking

[in] If true, this call doesn't return until the file has been received.

progressCallback

[in] Called periodically with progress updates. Can be NULL.

completedCallback

[in] Called when the file has been received. Can be NULL.

param

[in] Optional free-format user data for use by the callback

Remarks

This function is used to download the contents of a web page directly to disk. The application supplies the path and filename at which to save the response.

Use `ghttpSave` for a simpler version of this function.

Section Reference: [Gamespy HTTP SDK](#)

See Also: [ghttpGetEx](#), [ghttpSave](#), [ghttpStreamEx](#), [ghttpHeadEx](#), [ghttpPostEx](#)

ghttpSetMaxRecvTime

Used to throttle based on time, not on bandwidth.

```
void ghttpSetMaxRecvTime(  
    GHTTPRequest request,  
    gsi_time maxRecvTime );
```

Routine	Required Header	Distribution
ghttpSetMaxRecvTime	<ghttp.h>	SDKZIP

Parameters

request

[in] A valid request object

maxRecvTime

[in] Maximum receive time

Remarks

Prevents recv-loop blocking on ultrafast connections without directly limiting transfer rate.

Section Reference: [Gamespy HTTP SDK](#)

ghttpSetProxy

Sets a proxy server address.

```
GHTTPBool ghttpSetProxy(  
    const char * server );
```

Routine	Required Header	Distribution
<code>ghttpSetProxy</code>	<code><ghttp.h></code>	SDKZIP

Return Value

GHTTPFalse if the server format is invalid.

Parameters

server

[in] The address of the proxy server.

Remarks

The address must be of the form "<server>[:port]". If port is omitted, 80 will be used.

If server is NULL or "", no proxy server will be used. This should not be called while there are any current requests.

Section Reference: [Gamespy HTTP SDK](#)

See Also: [ghttpSetRequestProxy](#)

ghttpSetRequestProxy

Sets a proxy server for a specific request.

```
GHTTPBool ghttpSetRequestProxy(  
    GHTTPRequest request,  
    const char * server );
```

Routine	Required Header	Distribution
ghttpSetRequestProxy	<ghttp.h>	SDKZIP

Return Value

GHTTPFalse if the server format is invalid or the request is invalid.

Parameters

request

[in] A valid request object

server

[in] The address of the proxy server.

Remarks

The address must be of the form "<server>[:port]". If port is omitted, 80 will be used.

If server is NULL or "", no proxy server will be used. This should not be called while there are any current requests.

Section Reference: [Gamespy HTTP SDK](#)

See Also: [ghttpSetRequestProxy](#)

ghttpSetThrottle

Used to start/stop throttling an existing connection.

```
void ghttpSetThrottle(  
    GHTTPRequest request,  
    GHTTPBool throttle );
```

Routine	Required Header	Distribution
ghttpSetThrottle	<ghttp.h>	SDKZIP

Parameters

request

[in] A valid request object

throttle

[in] True or false to enable or disable throttling.

Remarks

This may not be as efficient as starting a request with the desired setting.

Section Reference: [Gamespy HTTP SDK](#)

See Also: [ghttpThrottleSettings](#)

ghttpStartup

Initialize the HTTP SDK.

void ghttpStartup();

Routine	Required Header	Distribution
ghttpStartup	<ghttp.h>	SDKZIP

Remarks

Startup/Cleanup is reference counted, so always call **ghttpStartup()** and **ghttpCleanup()** in pairs.

Section Reference: [Gamespy HTTP SDK](#)

See Also: [ghttpCleanup](#)

ghttpStream

Make a HTTP GET request and stream in the response without saving it in memory.

```
GHTTPRequest ghttpStream(  
    const gsi_char * URL,  
    GHTTPBool blocking,  
    ghttpProgressCallback progressCallback,  
    ghttpCompletedCallback completedCallback,  
    void * param );
```

Routine	Required Header	Distribution
ghttpStream	<ghttp.h>	SDKZIP

Return Value

If less than 0, the request failed and this is a `GHTTPRequestError` value. Otherwise it identifies the request.

Parameters

URL

[in] URL

blocking

[in] If true, this call doesn't return until the file has finished streaming.

progressCallback

[in] Called whenever new data is received. Can be NULL.

completedCallback

[in] Called when the file has finished streaming. Can be NULL.

param

[in] Optional free-format user data for use by the callback

Remarks

This function is used to stream in the contents of a web page. The response body is not stored in memory or to disk. It is only passed to the progressCallback as it is received, and the application can do what it wants with the data.

Use **ghttpStreamEx** for extra optional parameters.

Section Reference: [Gamespy HTTP SDK](#)

See Also: [ghttpGet](#), [ghttpSave](#), [ghttpStreamEx](#), [ghttpHead](#), [ghttpPost](#)

ghttpStreamEx

Make a HTTP GET request and stream in the response without saving it in memory.

```
GHTTPRequest ghttpStreamEx(  
    const gsi_char * URL,  
    const gsi_char * headers,  
    GHTTPPost post,  
    GHTTPBool throttle,  
    GHTTPBool blocking,  
    ghttpProgressCallback progressCallback,  
    ghttpCompletedCallback completedCallback,  
    void * param );
```

Routine	Required Header	Distribution
ghttpStreamEx	<ghttp.h>	SDKZIP

Return Value

If less than 0, the request failed and this is a `GHTTPRequestError` value. Otherwise it identifies the request.

Parameters

URL

[in] URL

headers

[in] Optional headers to pass with the request. Can be NULL or "".

post

[in] Optional data to be posted. Can be NULL.

throttle

[in] If true, throttle this connection's download speed.

blocking

[in] If true, this call doesn't return until the file has finished streaming.

progressCallback

[in] Called whenever new data is received. Can be NULL.

completedCallback

[in] Called when the file has finished streaming. Can be NULL.

param

[in] Optional free-format user data for use by the callback

Remarks

This function is used to stream in the contents of a web page. The response body is not stored in memory or to disk. It is only passed to the progressCallback as it is received, and the application can do what it wants with the data.

Use ghttpStream for a simpler version of this function.

Section Reference: [Gamespy HTTP SDK](#)

See Also: [ghttpGetEx](#), [ghttpSaveEx](#), [ghttpStream](#), [ghttpHeadEx](#), [ghttpPostEx](#)

ghttpThink

Processes all current http requests.

void ghttpThink();

Routine	Required Header	Distribution
ghttpThink	<ghttp.h>	SDKZIP

Remarks

Any application that uses GHTTP in non-blocking mode (sets the blocking parameter to GHTTPFalse) needs to call **ghttpThink** to let the library do any necessary processing. This call will process any current requests and call any callbacks if necessary. It will typically be called in the application's main loop. While it can be called as little as a few times a second, it should be called closer to 10-20 times a second. If downloading larger files, it may be desirable to call it even more often, to ensure that incoming buffers are emptied to make room for more incoming data.

Threads note: Making GHTTP requests concurrently from multiple threads is currently only supported under Win32. When using GHTTP from multiple threads, instead of calling **ghttpThink**, use `ghttpRequestThink` for each individual request. This allows that request's callback to be called from within the same thread in which it was started.

Section Reference: [Gamespy HTTP SDK](#)

See Also: [ghttpRequestThink](#)

ghttpThrottleSettings

Used to adjust the throttle settings.

```
void ghttpThrottleSettings(  
    int bufferSize,  
    gsi_time timeDelay );
```

Routine	Required Header	Distribution
ghttpThrottleSettings	<ghttp.h>	SDKZIP

Parameters

bufferSize

[in] The number of bytes to get each receive.

timeDelay

[in] How often to receive data, in milliseconds.

Remarks

The throttle settings affect any request initiated with throttling, or for which throttling is alter enabled with `ghttpSetThrottle`.

Section Reference: [Gamespy HTTP SDK](#)

See Also: [ghttpSetThrottle](#)

HTTP SDK Callbacks

[ghttpCompletedCallback](#)

Called when the entire file has been received.

[ghttpPostCallback](#)

Called during requests to let the app know how much of the post data has been uploaded.

[ghttpProgressCallback](#)

Called with updates on the current state of the request.

ghttpCompletedCallback

Called when the entire file has been received.

```
typedef GHTTPBool (*ghttpCompletedCallback)(  
    GHTTPRequest request,  
    GHTTPResult result,  
    char *buffer,  
    GHTTPByteCount bufferLen,  
    void *param );
```

Routine	Required Header	Distribution
ghttpCompletedCallback	<ghttp.h>	SDKZIP

Return Value

If `ghttpGetFile[Ex]` was used, return `true` to have the buffer freed, `false` if the app will free the buffer.

Parameters

request

[in] A valid request object

result

[in] The result (success or an error).

buffer

[in] The file's bytes (only valid if ghttpGetFile[Ex] was used).

bufferLen

[in] The file's length.

param

[in] Optional free-format user data for use by the callback

Remarks

If `ghttpStreamFileEx` or `ghttpSaveFile[Ex]` was used, `buffer` is `NULL`, `bufferLen` is the number of bytes in the file, and the return value is ignored.

If `ghttpGetFile[Ex]` was used, return `true` to have the buffer freed, `false` if the app will free the buffer. If `true`, the buffer cannot be accessed once the callback returns. If `false`, the app can use the buffer even after this call returns, but must free it at some later point. There will always be a file, even if there was an error, although for errors it may be an empty file.

Section Reference: [Gamespy HTTP SDK](#)

See Also: [ghttpGet](#), [ghttpGetEx](#), [ghttpSave](#), [ghttpSaveEx](#), [ghttpStream](#), [ghttpStreamEx](#), [ghttpHead](#), [ghttpHeadEx](#), [ghttpPost](#), [ghttpPostEx](#)

ghttpPostCallback

Called during requests to let the app know how much of the post data has been uploaded.

```
typedef void (*ghttpPostCallback)(  
    GHTTPRequest request,  
    int bytesPosted,  
    int totalBytes,  
    int objectsPosted,  
    int totalObjects,  
    void * param );
```

Routine	Required Header	Distribution
ghttpPostCallback	<ghttp.h>	SDKZIP

Parameters

request

[in] A valid request object

bytesPosted

[in] The number of bytes of data posted so far.

totalBytes

[in] The total number of bytes being posted.

objectsPosted

[in] The total number of data objects uploaded so far.

totalObjects

[in] The total number of data objects to upload.

param

[in] Optional free-format user data for use by the callback

Section Reference: [Gamespy HTTP SDK](#)

See Also: [ghttpNewPost](#), [ghttpPostSetCallback](#)

ghttpProgressCallback

Called with updates on the current state of the request.

```
typedef void (*ghttpProgressCallback)(  
    GHTTPRequest request,  
    GHTTPState state,  
    const char * buffer,  
    GHTTPByteCount bufferLen,  
    GHTTPByteCount bytesReceived,  
    GHTTPByteCount totalSize,  
    void * param );
```

Routine	Required Header	Distribution
ghttpProgressCallback	<ghttp.h>	SDKZIP

Parameters

request

[in] A valid request object.

state

[in] The current state of the request.

buffer

[in] The file's bytes so far, NULL if state < GHTTPReceivingFile.

bufferLen

[in] The number of bytes in the buffer, 0 if state < GHTTPReceivingFile.

bytesReceived

[in] The total number of bytes received, 0 if state < GHTTPReceivingFile.

totalSize

[in] The total size of the file, -1 if unknown.

param

[in] Optional free-format user data for use by the callback.

Remarks

The buffer should not be accessed once this callback returns.

If `ghttpGetFile[Ex]` was used, `buffer` contains all of the data that has been received so far, and `bufferSize` is the total number of bytes received.

If `ghttpSaveFile[Ex]` was used, `buffer` only contains the most recent data that has been received. This same data is saved to the file. The buffer will not be valid after this callback returns.

If `ghttpStreamFileEx` was used, `buffer` only contains the most recent data that has been received. This data will be lost once the callback returns, and should be copied if it needs to be saved. `bufferSize` is the number of bytes in the current block of data.

Section Reference: [Gamespy HTTP SDK](#)

See Also: [ghttpGetEx](#), [ghttpSaveEx](#), [ghttpStream](#), [ghttpStreamEx](#), [ghttpHeadEx](#), [ghttpPostEx](#)

HTTP SDK Enumerations

GHTTPBool	Standard Boolean.
GHTTPRequestError	Possible Error values returned from GHTTP functions.
GHTTPResult	The result of an HTTP request.
GHTTPState	The current state of an HTTP request.

GHTTPBool

Standard Boolean.

```
typedef enum  
{  
    GHTTPFalse,  
    GHTTPTrue  
} GHTTPBool;
```

Constants

GHTTPFalse
False.

GHTTPTrue
True.

Section Reference: [Gamespy HTTP SDK](#)

GHTTPRequestError

Possible Error values returned from GHTTP functions.

```
typedef enum  
{  
    GHTTPErrorStart,  
    GHTTPFailedToOpenFile,  
    GHTTPInvalidPost,  
    GHTTPInsufficientMemory,  
    GHTTPInvalidFileName,  
    GHTTPInvalidBufferSize,  
    GHTTPInvalidURL,  
    GHTTPUnspecifiedError  
} GHTTPRequestError;
```

Constants

GHTTPFailedToOpenFile
Failed to open file.

GHTTPInvalidPost
Invalid post.

GHTTPInsufficientMemory
Insufficient memory.

GHTTPInvalidFileName
Invalid filename.

GHTTPInvalidBufferSize
Invalid buffer size.

GHTTPInvalidURL
Invalid URL.

GHTTPUnspecifiedError
Unspecified error.

Section Reference: [Gamespy HTTP SDK](#)

GHTTPResult

The result of an HTTP request.

```
typedef enum  
{  
    GHTTPSuccess,  
    GHTTPOutOfMemory,  
    GHTTPBufferOverflow,  
    GHTTPParseURLFailed,  
    GHTTPHostLookupFailed,  
    GHTTPSocketFailed,  
    GHTTPConnectFailed,  
    GHTTPBadResponse,  
    GHTTPRequestRejected,  
    GHTTPUnauthorized,  
    GHTTPForbidden,  
    GHTTPFileNotFound,  
    GHTTPServerError,  
    GHTTPFileWriteFailed,  
    GHTTPFileReadFailed,  
    GHTTPFileIncomplete,  
    GHTTPFileTooBig  
} GHTTPResult;
```

Constants

GHTTPSuccess

Successfully retrieved file.

GHTTPOutOfMemory

A memory allocation failed.

GHTTPBufferOverflow

The user-supplied buffer was too small to hold the file.

GHTTPParseURLFailed

There was an error parsing the URL.

GHTTPHostLookupFailed

Failed looking up the hostname.

GHTTPSocketFailed

Failed to create/initialize/read/write a socket.

GHTTPConnectFailed

Failed connecting to the HTTP server.

GHTTPBadResponse

Error understanding a response from the server.

GHTTPRequestRejected

The request has been rejected by the server.

GHTTPUnauthorized

Not authorized to get the file.

GHTTPForbidden

The server has refused to send the file.

GHTTPFileNotFound

Failed to find the file on the server.

GHTTPServerError

The server has encountered an internal error.

GHTTPFileWriteFailed

An error occurred writing to the local file (for `ghttpSaveFile[Ex]`).

GHTTPFileReadFailed

There was an error reading from a local file (for posting files from

disk).

GHTTPFileIncomplete

Download started but was interrupted. Only reported if file size is known.

GHTTPFileToBig

The file is too big to be downloaded (size exceeds range of internal data types).

Section Reference: [Gamespy HTTP SDK](#)

GHTTPState

The current state of an HTTP request.

```
typedef enum  
{  
    GHTTPSocketInit,  
    GHTTPHostLookup,  
    GHTTPLookupPending,  
    GHTTPConnecting,  
    GHTTPSendingRequest,  
    GHTTPPosting,  
    GHTTPWaiting,  
    GHTTPReceivingStatus,  
    GHTTPReceivingHeaders,  
    GHTTPReceivingFile  
} GHTTPState;
```

Constants

GHTTPSocketInit

Startup socket.

GHTTPhostLookup

Begin resolving hostname to IP.

GHTTPLookupPending

Waiting for hostname to resolve (non-blocking).

GHTTPConnecting

Waiting for socket connect to complete.

GHTTPSendingRequest

Sending the request.

GHTTPPosting

Posting data (skipped if not posting).

GHTTPWaiting

Waiting for a response.

GHTTPReceivingStatus

Receiving the response status.

GHTTPReceivingHeaders

Receiving the headers.

GHTTPReceivingFile

Receiving the file.

Section Reference: [Gamespy HTTP SDK](#)

NAT Negotiation SDK

Overview

The GameSpy NAT Negotiation SDK interacts with GameSpy's NAT Negotiation server to allow hosting of multiplayer games by users behind NAT and firewall devices. Typically, a user behind a NAT or firewall device cannot host multiplayer games because the device will block incoming connections from outside users. GameSpy's NAT Negotiation technology allows two users, one or both of whom are behind a NAT device, to open a clear UDP channel directly between the users.

For background information on the networking challenges posed by NAT and Firewall devices, see the appendix of the Query & Reporting 2 documentation entitled "NAT and Firewall Support". This document assumes you are familiar with the terminology and issues discussed in that document.

GameSpy's NAT Negotiation technology uses a method known as "Port Guessing" to attempt to discern future port mapping information for two users based on their connections to the NAT Negotiation server. Once this mapping information is determined, the server exchanges the information with the users, and they connect to each other directly (note: the term "connect" in this document is understood to mean the establishment a clear, two-way channel between the users, since UDP is in reality a connection-less protocol).

Note that the NAT Negotiation SDK does not make any distinction between the "client" who is connecting to a "server" (or "host"), however this document will use those terms for clarity, and because the other SDKs involved do make that distinction.

The NAT Negotiation SDK itself is very simple - two users who want to be connected to each other have a shared "cookie" value that the NAT Negotiation server uses to match the users up.

The NAT Negotiation SDK has no limit to the number of users that can be connected together, but each channel between two users must be

independently established.

In order for a client to connect to a server behind a NAT, both the client and the server must know a shared cookie value. Because the server is behind a NAT, the client cannot communicate this value to the server directly (if it could, there would be no need for the NAT negotiation step!). To communicate indirectly to the server, the client uses the Peer or ServerBrowsing SDK (depending on which of the two SDKs the game has implemented) to send a connection request to the GameSpy Master Server. The GameSpy Master Server then sends this connection request directly to the game server.

Because the game server is sending heartbeats to the Master Server (via the Peer or Query & Reporting 2 SDK), the Master Server has an open channel to communicate back to the game server. It uses this channel to pass the connection cookie back to the server. The server and client are now both aware of the cookie, and can use the NAT Negotiation SDK to establish a connection directly between them.

Supported Network Models

The NAT Negotiation SDK supports the establishment of UDP connections between two users, with one or both behind a NAT. For games using a dedicated server model, a number of clients will generally connect and disconnect from the server over time. For a Peer-to-Peer model, the game may have a centralized host that everyone connects to, or may require a distinct connection between each set of Peers. Any of these models will function using the NAT Negotiation SDK. The only limitations imposed are that TCP networking is not supported, and that the game must have direct access to the underlying UDP "SOCKET" - thus abstracted networking layers such as DirectPlay that hide the underlying sockets may not be compatible.

Some games use a single UDP socket for all communications to all clients that are connected to a server. Other games use a distinct socket for each 2-party connection. Both models are compatible with the NAT Negotiation SDK. For games that use a single socket for all clients, this socket is allocated by the game, and the game reads messages off of it and passes them into the NAT Negotiation SDK as needed for processing. For games that require a single UDP socket for each connection, the NAT Negotiation SDK allocates this socket as part of the negotiation process and passes it back to the game when negotiation is complete.

Note that if you are sharing a socket with the Query & Reporting (or Peer SDK) for server queries, this same socket can be shared for all client communications and NAT Negotiation as well. You will simply need to look at the packets as they come in and determine whether to pass them to the Q&R SDK or to the NAT Negotiation SDK (or process them as game networking packets).

File Manifest

The following files should be included with this package. If any of the files are missing, please [contact devsupport@gamespy.com](mailto:devsupport@gamespy.com).

File	Description
<i>natneg.c</i>	NAT Negotiation SDK source
<i>natneg.h</i> your source	NAT Negotiation MAIN HEADER - include this in
<i>nninternal.h</i>	Internal structures and defines
<i>\simpletest\</i>	Simple connection sample

In addition, to build the SDK and samples, you will need to separately download the GameSpy "common code" package, which includes the shared SDK code used by this SDK and others.

When extracting this package, make sure you preserve the directory tree in order to ensure that the code builds correctly.

Implementation

Step 1: Server-side changes

For the server to be able to host behind a NAT, several changes are needed to the basic implementation of the Query & Reporting 2 or Peer SDK (follow the instructions for the SDK you are using).

These SDKs report the availability of your server to our Master Server. Normally, our Master Server does not allow games to be hosted from behind a NAT, as this would prevent outside clients from being able to connect to them. However, because you are using the NAT Negotiation SDK, your game client will be able to connect to servers behind a NAT and our Master Server needs to be aware of this. To indicate this support to our Master Server simply pass: "1" to the "[natnegotiate](#)" parameter of `qr2_init()` [Q&R 2 SDK] or "[PEERTrue](#)" to the "[natNegotiate](#)" parameter of `peerSetTitle()` [Peer SDK].

Note that you can set that parameter to true whether or not the current user hosting a game is behind a NAT. The backend will determine whether the user is behind a NAT automatically and inform clients whether they can connect directly or need to attempt NAT Negotiation. You simply need to set the parameter to indicate that you support NAT Negotiation as an option.

If a client wishes to connect to the server via NAT Negotiation, it will send a cookie value to the master server, which will forward it directly to your server. The Peer or QR2 SDK will indicate this request via a callback, as described below.

Query & Reporting 2 SDK

You need to create a callback function that will be called when a NAT Negotiate request comes in. The prototype for this callback is as follows:

```
typedef void (*qr2_natnegcallback_t)(int cookie, voi
```

The cookie value is the value passed from the connecting client, which will be used with the NAT Negotiate SDK.

Once you have created your callback, you need to register it with:

```
void qr2_register_natneg_callback(qr2_t qrec, qr2_na
```

You should generally call this immediately after `qr2_init()`.

Peer SDK

You need to create a callback function that will be called when a NAT Negotiate request comes in. The prototype for this callback is as follows:

```
typedef void (* peerQRNatNegotiateCallback)(PEER pee
```

The cookie value is the value passed from the connecting client, which will be used with the NAT Negotiate SDK.

Pass this function in as part of the `PEERCallbacks` structure in `peerInitialize`.

Step 2: Client-side changes

On your client, which is doing matchmaking via either the ServerBrowsing or Peer SDK, you need to take special steps when connecting to a server behind a NAT versus a directly accessible server.

When a client selects the server they wish to connect to, the first step is to determine whether the server is behind a NAT, and if so, whether it is behind the same NAT as the client. This is important because most NATs will not allow inside clients to connect to inside servers via the "public" IP address on the NAT - the client must connect to the server directly via the private IP address.

To check this, call:

```
SBBool SBServerHasPrivateAddress(SBServer server);
```

... to determine whether the server has a private Address. If it does, you should compare the public address of the server with the public address of your current client to determine if they match.

You can determine the server's public address with:

```
unsigned int SBServerGetPublicInetAddress(SBServer s
```

... you will compare this value to the value returned from:

```
unsigned int ServerBrowserGetMyPublicIPAddr(ServerBr
```

... Or ...

```
unsigned int peerGetPublicIP(); [Peer SDK]
```

If the public address of the server and your client match, then you know both are behind the same NAT, and you should connect via the "Internal" address of the server. You can now proceed to connect directly to the server via the Private address (obtained with [SBServerGetPrivateAddress](#) or [SBServerGetPrivateInetAddress](#), depending on the preferred format for your game).

If the server is not behind the same NAT as your client, you need to determine whether to connect to the server directly, or initiate NAT Negotiation. Direct connections can be used if the server is either not behind a NAT, or is behind a promiscuous NAT and your game uses shared sockets.

To determine whether a direct connection can be made, use the function:

```
SBBool SBServerDirectConnect(SBServer server);
```

If the server supports direct connection, you can proceed to connect to the server directly using the Public address (obtain via [SBServerGetPublicAddress](#) or [SBServerGetPublicInetAddress](#)).

If the server does not support direct connection, you will need to proceed with NAT Negotiation.

On the client, you need to generate a random "cookie" value to send to the server. This cookie is used by the NAT Negotiation server to match up requests for connection. Simply generate a random 32-bit integer and call: [ServerBrowserSendNatNegotiateCookieToServer\(\)](#) [ServerBrowsing SDK] or [peerSendNatNegotiateCookie\(\)](#) [Peer SDK]. Pass in the public IP address and public query port for the server (obtain via [SBServerGetPublicAddress](#) + [SBServerGetPublicQueryPort](#)). This will send the cookie value to the server and trigger the NAT Negotiation callback as described in Step 1.

Once you have sent the cookie, you should proceed with NAT Negotiation as described below.

Step 3: Initiate NAT Negotiation

On the client, immediately after sending the cookie value to the server, you should begin the NAT Negotiation process. To do this, simply call one of the BeginNegotiation functions below:

```
NegotiateError NNBeginNegotiation(int cookie, int cl
```

```
NegotiateError NNBeginNegotiationWithSocket(SOCKET g
```

Which version you use depends on how your game networking is set up. If you want to use the same socket for all client connections, and manage incoming data on the socket yourself, use the "NNBeginNegotiationWithSocket" function and pass in the socket you want to use. If you want the NAT Negotiation SDK to allocate a new socket for each connection, use the "NNBeginNegotiation" function. The

“cookie” parameter is the cookie value you generated and sent to the server. The “clientindex” parameter is simply 0 for the client, and “1” for the server (or reversed – it doesn’t matter as long as one of them is 0, and the other is 1). The progress function will be called to update you on the negotiation progress and the completed callback will be called when the negotiation is complete. The userdata will be passed into your callback functions. See below for a description of the individual callbacks.

On your server, the process is much the same. When your NAT Negotiation callback is called, you want to begin negotiation using the cookie value provided. Simply call the appropriate NNBeginNegotiation function as described above.

Step 4: NAT Negotiation Callbacks

The NAT Negotiation SDK requires two callbacks - a progress callback that gets called as negotiation is proceeding, and a completed callback when negotiation is complete.

The progress function prototype is:

```
typedef void (*NegotiateProgressFunc)(NegotiateState
```

The two times you will get a progress notification is when the NAT Negotiation server acknowledges your connection request (`ns_initack`), and when the guessed port data has been received from the NAT Negotiation server and direct negotiation with the other client is in progress (`ns_connectping`).

The completed function prototype is:

```
typedef void (*NegotiateCompletedFunc)(NegotiateResu
```

`result` will indicate the result of the negotiation attempt. Possible values are:

`nr_success`

Successful negotiation, an open channel has now been

established.

`nr_deadbeatpartner`

Partner did not register with the NAT Negotiation Server.

`nr_inittimeout`

Unable to communicate with NAT Negotiation Server

`nr_unknownerror`

NAT Negotiation server indicated an unknown error condition

`gamesocket` is the socket you should use to continue communications with the client. If you used `NNBeginNegotiationWithSocket` then this will be the socket you passed in originally. Otherwise it will be a new socket allocated by the NAT Negotiation SDK.

`remoteaddr` is the remote address and port you should use to communicate with the new client. Make sure you copy this structure off before the callback returns.

`userdata` is for your own use.

Once your completed function is called, you can begin sending data to the other client immediately using the socket and address provided.

Step 5: Thinking and Processing Incoming Data

After you've begun negotiation, you need to call the `NNThink()` function on regular intervals (recommended: 100 ms) to process the connection. You may call `NNThink()` when no negotiations are in progress as well - it will simply return immediately.

If you are using a shared game socket for all communications (`NNBeginNegotiationWithSocket`) then in addition to calling `NNThink()` you will need to look for NAT Negotiation packets arriving on that socket and pass them into the SDK. The SDK considers your game the "owner" of that socket and will not try to read any data from it directly. Simply pass the data, length, and received address obtained from "recvfrom" to:

```
void NNProcessData(char *data, int len, struct socka
```

To identify NAT Negotiation packets, you can use the 6 magic bytes that are used at the beginning of every packet. These are defined in *natneg.h* starting with `NN_MAGIC_0`.

Note that even after negotiation is complete and the completed callback is called, you need to continue looking for incoming NAT Negotiation packets on that socket for at least 5 seconds and continue to pass them to `NNProcessData`. Due to the unreliable nature of UDP, some packets may get sent even after one side of the connection has determined that a connection is established (due to dropped packets, etc). You should also call `NNThink()` during this time period as well.

If you want to cancel a Negotiation in progress, you can do so at any time by passing the cookie value to `NNCancel()`.

Step 6: Initiate Standard Game Networking

Both clients will receive the completed callback at about the same time. At this point you can commence normal network interaction between the clients using the sockets and addresses provided.

Step 7: Cleanup the Nat Negotiation SDK

Once you have finished negotiating, the internal SDK memory must be freed using `NNFreeNegotiatorList`. Calling this will NOT close the game sockets, you are free to continue game communications.

Appendix: Test Results

We have tested the NAT Negotiation SDK with a variety of hardware and software NAT devices, as outlined below. We will continue testing with new devices in the future to make sure we have the widest possible support, although the devices we have currently tested with represent most if not all of the NAT management schemes we are aware of, so most other devices are likely to be compatible.

Device	HW/SW	Port Mapping Scheme	Success	Comments
Dlink DI-604 Residential Gateway	HW	1:tuple	YES	
LinkSys Cable / DSL Router BEFSR41 v.1.34	HW	1:1 exact*	YES*	Has a bug in this version that reuses the same mapping for multiple clients if behind the same NAT using the same port.
LinkSys Cable / DSL Router BEFSR41 v.1.47	HW	1:1 exact / 1:1 port	YES	
SMC Barricade 7004VBR	HW	New Every Packet*	YES*	Has a bug that causes it to allocate a new port for every packet. Connection can be established, but incoming ports for every packet will be different.
USR Broadband	HW	1:1 exact / 1:tuple	YES*	Second client using same port behind NAT

Router 8000A					will get a random port allocation and may not be able to connect.
Belkin Wireless Router F5D6230	HW	1:1 exact / 1:tuple	YES*		Identical behavior to USR
Netgear Prosafe Firewall FR114	HW	1:1 per-ip	YES		
Windows 2000 ICS	SW	1:1 port (sometimes exact)	YES*		Uses unusual port allocation scheme. May result in bad port guess during rapid connections.
Sygate Home Network 4.2	SW	1:1 port	YES		
Coyote Linux / IP Chains	SW	1:tuple	YES		
Floppy FW (Linux / IPTables)	SW	1:1 exact	YES		

NAT Negotiation SDK Functions

NNBeginNegotiation	Starts the negotiation process.
NNBeginNegotiationWithSocket	Starts the negotiation process using the socket provided, which will be shared with the game.
NNCancel	Cancels a NAT Negotiation request in progress
NNFreeNegotiateList	De-allocates the memory used by for the negotiate list when you are done with NAT Negotiation.
NNProcessData	Processes data received from a shared socket.
NNStartNatDetection	Starts the NAT detection process.
NNThink	Processes any negotiation or NAT detection requests that are in progress.

NNBeginNegotiation

Starts the negotiation process.

```
NegotiateError NNBeginNegotiation(  
    int cookie,  
    int clientindex,  
    NegotiateProgressFunc progresscallback,  
    NegotiateCompletedFunc completedcallback,  
    void * userdata );
```

Routine	Required Header	Distribution
NNBeginNegotiation	<natneg.h>	SDKZIP

Return Value

ne_noerror if successful; otherwise one of the ne_error values. See Remarks for detail.

Parameters

cookie

[in] Shared cookie value that both players will use so that the NAT Negotiation Server can match them up.

clientindex

[in] One client must use clientindex 0, the other must use clientindex 1.

progresscallback

[in] Callback function that will be called as the state changes.

completedcallback

[in] Callback function that will be called when negotiation is complete.

userdata

[in] Pointer for your own use that will be passed into the callback functions.

Remarks

Possible errors that can be returned when starting a negotiation

ne_noerror: No error

ne_allocerror~~trong~~**>**: ~~Memory allocation failed~~

~~> ne_socketerror~~: Socket allocation failed

~~ne_dnserror~~: DNS lookup failed.

Section Reference: [Gamespy NAT Negotiation SDK](#)

See Also: [NNBeginNegotiationWithSocket](#)

NNBeginNegotiationWithSocket

Starts the negotiation process using the socket provided, which will be shared with the game.

```
NegotiateError NNBeginNegotiationWithSocket(  
    SOCKET gamesocket,  
    int cookie,  
    int clientindex,  
    NegotiateProgressFunc progresscallback,  
    NegotiateCompletedFunc completedcallback,  
    void * userdata );
```

Routine	Required Header	Distribution
NNBeginNegotiationWithSocket	<natneg.h>	SDKZIP

Return Value

Possible errors that can be returned when starting a negotiation

ne_noerror: No error

ne_allocerror: Memory allocation failed

ne_socketerror: Socket allocation failed

ne_dnserror: DNS lookup failed

Parameters

gamesocket

[in] The socket to be used to start the negotiation

cookie

[in] Shared cookie value that both players will use so that the NAT Negotiation Server can match them up.

clientindex

[in] One client must use clientindex 0, the other must use clientindex 1.

progresscallback

[in] Callback function that will be called as the state changes.

completedcallback

[in] Callback function that will be called when negotiation is complete.

userdata

[in] Pointer for your own use that will be passed into the callback functions.

Remarks

Incoming traffic is not processed automatically - you will need to read the data off the socket and pass NN packets to NNProcessData.

Section Reference: [Gamespy NAT Negotiation SDK](#)

See Also: [NNBeginNegotiation](#)

NNCancel

Cancels a NAT Negotiation request in progress.

```
void NNCancel(  
    int cookie );
```

Routine	Required Header	Distribution
NNCancel	<natneg.h>	SDKZIP

Parameters

cookie

[in] The cookie associated with this negotiation

Section Reference: [Gamespy NAT Negotiation SDK](#)

NNFreeNegotiateList

De-allocates the memory used by for the negotiate list when you are done with NAT Negotiation.

void NNFreeNegotiateList();

Routine	Required Header	Distribution
NNFreeNegotiateList	<natneg.h>	SDKZIP

Remarks

Once you have finished negotiating, the internal SDK memory must be freed using `NNFreeNegotiatorList`.

If any negotiations are outstanding this will cancel them. Calling this will NOT close the game sockets, you are free to continue game communications.

Section Reference: [Gamespy NAT Negotiation SDK](#)

NNProcessData

Processes data received from a shared socket.

```
void NNProcessData(  
    char * data,  
    int len,  
    struct sockaddr_in * fromaddr );
```

Routine	Required Header	Distribution
NNProcessData	<natneg.h>	SDKZIP

Parameters

data

[in] The data packets read from the gamesocket.

len

[in] Length of the data.

fromaddr

[in] The address from which the data packets came.

Remarks

When sharing a socket with the NAT Negotiation SDK, you must read incoming data and pass NN packets to **NNProcessData**.

Section Reference: [Gamespy NAT Negotiation SDK](#)

See Also: [NNBeginNegotiationWithSocket](#)

NNStartNatDetection

Starts the NAT detection process.

NegotiateError NNStartNatDetection(
NatDetectionResultsFunc resultscallback);

Routine	Required Header	Distribution
NNStartNatDetection	<natneg.h>	SDKZIP

Return Value

ne_noerror if successful; otherwise one of the ne_error values. See Remarks for detail.

Parameters

resultcallback

[in] Callback function that will be called when NAT detection is complete.

Remarks

Possible errors that can be returned when starting a negotiation

ne_noerror: No error

ne_socketerror: Socket allocation failed

ne_dnserror: DNS lookup failed.

Section Reference: [Gamespy NAT Negotiation SDK](#)

See Also: [NatDetectionResultsFunc](#), [NAT](#)

NNThink

Processes any negotiation or NAT detection requests that are in progress.

void NNThink();

Routine	Required Header	Distribution
NNThink	<natneg.h>	SDKZIP

Remarks

After you've begun a negotiation and/or NAT detection, you need to call the **NNThink** function on regular intervals (recommended: 100ms) to process the connection. You may call **NNThink** when no negotiations are in progress as well - it will simply return immediately.

Section Reference: [Gamespy NAT Negotiation SDK](#)

NAT Negotiation SDK Callbacks

[NatDetectionResultsFunc](#)

The callback that gets executed from NNStartNatDetection when the detection is complete.

[NegotiateCompletedFunc](#)

The callback that gets executed from NNBeginNegotiation when negotiation is complete.

[NegotiateProgressFunc](#)

The callback that gets executed from NNBeginNegotiation as negotiation is proceeding.

NatDetectionResultsFunc

The callback that gets executed from NNStartNatDetection when the detection is complete.

```
typedef void (*NatDetectionResultsFunc)(  
    gsi_bool success,  
    NAT nat );
```

Routine	Required Header	Distribution
NatDetectionResultsFunc	<natneg.h>	SDKZIP

Parameters

success

[in] if `gsi_true` the NAT detection was successful

nat

[in] When detection is successful, this contains the NAT device's properties.

Remarks

Once your detection callback function is called, check the success parameter. If it is `gsi_false`, then the detection could not be completed and should be retried. If it is `gsi_true`, then the `nat` parameter will contain the properties of the detected NAT device.

Section Reference: [Gamespy NAT Negotiation SDK](#)

See Also: [NNStartNatDetection](#), [NAT](#)

NegotiateCompletedFunc

The callback that gets executed from NNBeginNegotiation when negotiation is complete.

```
typedef void (*NegotiateCompletedFunc)(  
    NegotiateResult result,  
    SOCKET gamesocket,  
    struct sockaddr_in * remoteaddr,  
    void * userdata );
```

Routine	Required Header	Distribution
NegotiateCompletedFunc	<natneg.h>	SDKZIP

Parameters

result

[in] Indicates the result of the negotiation attempt.

gamesocket

[in] The socket you should use to continue communications with the client.

remoteaddr

[in] The remote address and port you should use to communicate with the new client.

userdata

[in] Data for your own use.

Remarks

Once your completed function is called, you can begin sending data to the other client immediately using the socket and address provided.

Possible values for the value of the result parameter are:

nr_success

Successful negotiation, an open channel has now been established.

nr_deadbeatpartner

Partner did not register with the NAT Negotiation Server.

nr_inittimeout

Unable to communicate with NAT Negotiation Server

nr_pingtimeout

Unable to communicate directly with partner

nr_unknownerror

NAT Negotiation server indicated an unknown error condition

If you used `NNBeginNegotiationWithSocket` then the socket parameter will be the socket you passed in originally. Otherwise it will be a new socket allocated by the NAT Negotiation SDK.

Make sure you copy the `remoteaddr` structure before the callback returns.

Section Reference: [Gamespy NAT Negotiation SDK](#)

See Also: [NNBeginNegotiation](#)

NegotiateProgressFunc

The callback that gets executed from NNBeginNegotiation as negotiation is proceeding.

```
typedef void (*NegotiateProgressFunc)(  
    NegotiateState state,  
    void * userdata );
```

Routine	Required Header	Distribution
NegotiateProgressFunc	<natneg.h>	SDKZIP

Parameters

state

[in] The state of the negotiation at the time of notification.

userdata

[in] Data for your own use.

Remarks

The two times you will get a progress notification is when the NAT Negotiation server acknowledges your connection request (ns_initack), and when the guessed port data has been received from the NAT Negotiation server and direct negotiation with the other client is in progress (ns_connectping).

Section Reference: [Gamespy NAT Negotiation SDK](#)

See Also: [NNBeginNegotiation](#)

NAT Negotiation SDK Structures

[AddressMapping](#)

Internal and external address pairing for an observed network address translation.

[NAT](#)

The result of a NAT detection. Upon successful completion of a detection, this will contain as many properties of the NAT as could be determined.

AddressMapping

Internal and external address pairing for an observed network address translation.

```
typedef struct  
{  
    unsigned int privateIp;  
    unsigned short privatePort;  
    unsigned int publicIp;  
    unsigned short publicPort;  
} AddressMapping;
```

Members

privateIp

Internal IP address.

privatePort

Internal port number.

publicIp

External IP address.

publicPort

External port number.

Section Reference: [Gamespy NAT Negotiation SDK](#)

See Also: [NAT](#)

NAT

The result of a NAT detection. Upon successful completion of a detection, this will contain as many properties of the NAT as could be determined.

```
typedef struct  
{  
    char brand[32];  
    char model[32];  
    char firmware[64];  
    gsi_bool ipRestricted;  
    gsi_bool portRestricted;  
    NatPromiscuity promiscuity;  
    NatType natType;  
    NatMappingScheme mappingScheme;  
    AddressMapping mappings[4];  
    gsi_bool qr2Compatible;  
} NAT;
```

Members

brand

NAT device brand/vendor (not currently used).

model

NAT device model name/number (not currently used).

firmware

NAT device brand/vendor (not currently used).

ipRestricted

gsi_true if the NAT drops packets from unsolicited IP addresses.

portRestricted

gsi_true if the NAT drops packets from unsolicited ports.

promiscuity

The type of promiscuity the NAT allows.

natType

The type of NAT as defined by RFC2663.

mappingScheme

The type of port mapping/allocation scheme used by the NAT.

mappings

Port mappings observed during the detection process.

qr2Compatible

gsi_true if the NAT is compatible with QR2.

Section Reference: [Gamespy NAT Negotiation SDK](#)

See Also: [NNStartNatDetection](#), [NatDetectionResultsFunc](#), [NatType](#), [NatMappingScheme](#), [NatPromiscuity](#), [AddressMapping](#)

NAT Negotiation SDK Enumerations

NatMappingScheme	Common NAT port allocation schemes.
NatPromiscuity	The level of promiscuity (allowed traffic) for a NAT device.
NatType	NAT types based on RFC2663.
NegotiateError	Possible error values that can be returned when starting a negotiation.
NegotiateResult	Possible results of the negotiation.
NegotiateState	Possible states for the SDK. The two you will be notified for are ns_initack and ns_connectping.

NatMappingScheme

Common NAT port allocation schemes.

```
typedef enum  
{  
    unrecognized,  
    private_as_public,  
    consistent_port,  
    incremental,  
    mixed  
} NatMappingScheme;
```

Constants

unrecognized

The mapping scheme is not recognized. This could also mean it is a random scheme.

private_as_public

The public port is the same as the private port.

consistent_port

The same public port is being used for all requests from the same private port.

incremental

Each new mapped port is an increment over the previous one.

mixed

A mixed mapping scheme is being used.

Section Reference: [Gamespy NAT Negotiation SDK](#)

See Also: [NAT](#)

NatPromiscuity

The level of promiscuity (allowed traffic) for a NAT device.

```
typedef enum  
{  
    promiscuous,  
    not_promiscuous,  
    port_promiscuous,  
    ip_promiscuous,  
    promiscuity_not_applicable  
} NatPromiscuity;
```

Constants

promiscuous

All unsolicited traffic allowed.

not_promiscuous

No unsolicited traffic allowed.

port_promiscuous

Traffic from the same IP on a different port allowed.

ip_promiscuous

Traffic from a different IP allowed.

promiscuity_not_applicable

Does not apply to the type of NAT device.

Section Reference: [Gamespy NAT Negotiation SDK](#)

See Also: [NAT](#)

NatType

NAT types based on RFC2663.

```
typedef enum  
{  
    no_nat,  
    firewall_only,  
    full_cone,  
    restricted_cone,  
    port_restricted_cone,  
    symmetric,  
    unknown  
} NatType;
```

Constants

no_nat

No network address translation.

firewall_only

No network address translation, but firewall may be present.

full_cone

Full Cone type network address translation.

restricted_cone

Restricted Cone type network address translation.

port_restricted_cone

Port Restricted Cone type network address translation.

symmetric

Symmetric type network address translation.

unknown

Unrecognized type of network address translation.

Section Reference: [Gamespy NAT Negotiation SDK](#)

See Also: [NAT](#)

NegotiateError

Possible error values that can be returned when starting a negotiation.

```
typedef enum  
{  
    ne_noerror,  
    ne_allocerror,  
    ne_socketerror,  
    ne_dnserror  
} NegotiateError;
```

Constants

ne_noerror

No error.

ne_allocerror

Memory allocation failed.

ne_socketerror

Socket allocation failed.

ne_dnserror

DNS lookup failed.

Section Reference: [Gamespy NAT Negotiation SDK](#)

See Also: [NNBeginNegotiation](#)

NegotiateResult

Possible results of the negotiation.

```
typedef enum  
{  
    nr_success,  
    nr_deadbeatpartner,  
    nr_inittimeout,  
    nr_pingtimeout,  
    nr_unknownerror,  
    nr_noresult  
} NegotiateResult;
```

Constants

nr_success

Successful negotiation, other parameters can be used to continue communications with the client.

nr_deadbeatpartner

Partner did not register with the NAT Negotiation Server.

nr_inittimeout

Unable to communicate with NAT Negotiation Server.

nr_pingtimeout

Unable to communicate with partner.

nr_unknownerror

NAT Negotiation server indicated an unknown error condition.

nr_noresult

Initial negotiation status before a result is determined.

Section Reference: [Gamespy NAT Negotiation SDK](#)

See Also: [NegotiateCompletedFunc](#)

NegotiateState

Possible states for the SDK. The two you will be notified for are ns_initack and ns_connectping.

```
typedef enum  
{  
    ns_initsent,  
    ns_initack,  
    ns_connectping,  
    ns_finished,  
    ns_canceled,  
    ns_reportsent,  
    ns_reportack  
} NegotiateState;
```

Constants

ns_initsent

Initial connection request has been sent to the server (internal).

ns_initack

NAT Negotiation server has acknowledged your connection request.

ns_connectping

Direct negotiation with the other client has started.

ns_finished

The negotiation process has completed (internal).

ns_canceled

The negotiation process has been canceled (internal).

ns_reportsent

Negotiation result report has been sent to the server (internal).

ns_reportack

NAT Negotiation server has acknowledged your result report (internal).

Section Reference: [Gamespy NAT Negotiation SDK](#)

See Also: [NegotiateProgressFunc](#)

Patching and Usage Analysis SDK

Overview

GameSpy's Patching SDK simplifies the process of determining when a patch is required for a game and delivering it to the user.

Unlike other patching systems that have complex scripting and file matching requirements, our SDK requires no changes to the way developers and publishers create patches, and instead focuses on patch identification and delivery. Developers can use existing patch creation tools to create and install their patch, and use the Patching SDK to manage delivery to users.

The SDK is supported by an easy-to-use web interface that developers can use to configure updates. Different updates can be delivered for different versions or distributions of the game. For example, you can have a small patch for version 1.1 to 1.2, and a larger patch to 1.2 for users that still have version 1.0. You can also have different patches based on distribution, so different languages or platforms can have their own set of patches. You can even set the current version on a per-distribution basis, since many games release a patch in one region before localized versions. You can also use distribution patching to allow internal or external beta testers to update to a new version before it's publicly available.

The SDK is also fully-compatible with manually downloading and installing patches - so you can make the patch files available on your web site or on a magazine cover CD, in addition to being available in-game.

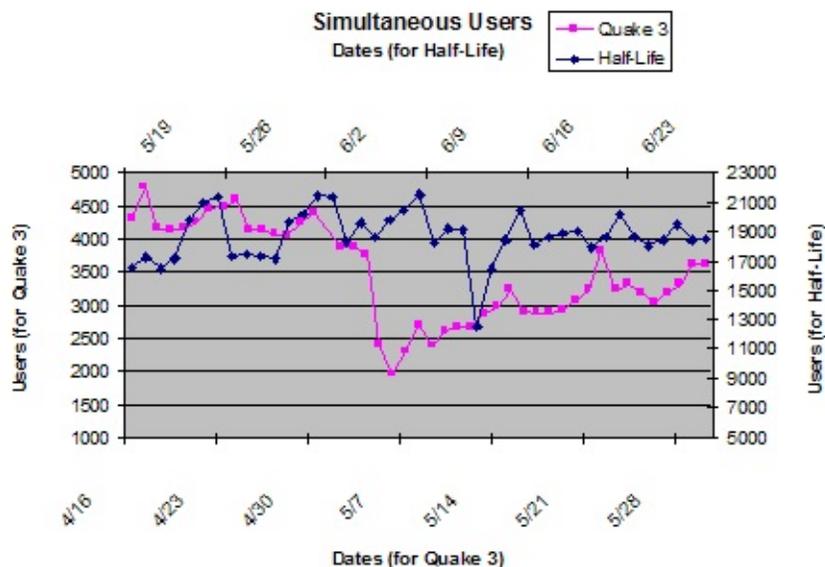
The Patching SDK and the backend that supports it have been used for over four years by GameSpy for updating our own software and has been found to be an extremely stable and reliable method of delivering updates to millions of our own users.

Benefits

Auto-patching of games has many potential benefits. Making sure patches are easy to find and install helps reduce customer service issues - when bugs are found and corrected, they fixes are easily distributed to all players, instead of relying on customer service operators directing users to web sites.

The ability to patch the game with upgrades or new content can help extend the life of a game online, and delivering this new content to all players assures the critical mass necessary for it to succeed.

We feel strongly that lack of auto-patching for multiplayer games can be a serious detriment to online play. To demonstrate this, we present below graphs from our master server of online play for two popular multiplayer games over a range of dates. During the date ranges charted on the graph, both games had a major patch released.



Quake 3, which does not have an auto-patching system (although it does have a patch notification system in-game), was averaging around 4200 simultaneous players before their patch on 5/4. After the patch, usage dropped sharply and it was more than a month before online play

returned to previous levels. There are undoubtedly large numbers of players that were never able to locate or install the patch, and thus never returned to play the game online.

In contrast, Half-Life, which has a combination internal/external auto-patching system, was averaging 19000 simultaneous players before their patch release on 6/8. After the release there is a short dip, but usage quickly returned to normal as players are automatically patched to the latest version.

INSERT IMAGE HERE, y0!

How It Works

User Perspective

First, here is an example of what a user might see in a game using the Patching SDK.

1. Player launches the game
2. After launching the game, the player is notified that a new version is available
3. The player is given the option to download and install the patch
4. After the download is complete, the game exits and the patch installer begins the patch installation process
5. Once the patch install is complete the game is re-launched and the user continues

Developer Perspective

To understand how the system works "under the covers" it is important to first understand what we classify as a unique version.

A version is identified by the combination of 3 identifiers:

productID

Each game is issued a unique productid by GameSpy for their game. Games on multiple platforms may be issued multiple productIDs.

distributionID

Different distributions can be patched with different patches. If you only have one distribution, simply use 0 for the distribution ID. Otherwise send us a list of the distributions you want patched separately and we'll send you distribution IDs for them.

versionUniqueID

This is a string that uniquely identifies a particular versions. This string can be anything you want, as long as it is different for each version you want to differentiate between. For example, it can be "1" then "2" then "3" or "1.0" then "1.01" then "1.1" or even

"version1" "next version" "third version". This string is not shown to users anywhere, any may be up to 30 characters long.

When the game is running and wants to check if a patch is available it first determines its identifiers. These identifiers are usually compiled in (for example, as defines), but can also be read from an external resource if its more convenient.

The game then calls `ptCheckForPatch()` with the identifiers, and a callback that will get called when the check is complete. This call can either be blocking or non-blocking. If done non-blocking, the function `ghttpThink()` must be called on a regular basis until the check is complete to poll for results.

The Patching SDK then contacts the GameSpy Patching Backend to determine whether an update is required for the user, and if so what patch they need to get.

To determine whether an update is required, the Patching Backend consults the **Current Version List** for the given productID. The Current Version List can have a different current version for each distribution, and is editable using the web interface as described below.

Once the Patching Backend determines the current version for the given productID and distributionID, it compares it to the versionUniqueID passed from the game to see if they match. If they do, the user has the current version and the backend notifies the SDK.

If the current version does not match, then the Patching Backend needs to determine what patch (if any) to send the user. Before checking the list of patches, it checks the **Known Version List** for the versionUniqueID passed from the game. If that uniqueID does not exist in the list of known versions, it adds it. If it does exist, it checks whether the "Internal Version" flag is set. If the "Internal Version" flag is set on this version reported from the game, then no patching is done (typically this is used for betas or other test versions, that are not technically the current version, but should not be patched to the current version either). This Known Version List can be edited to give versions descriptive names (that will be displayed to users when asked to patch) or mark versions as

Internal using the web interface described below.

Next the Patching Backend checks the **Patch Information List** to locate a patch that will get this version to the latest version. The Patch Information List (which is editable using the web interfaces described below) contains a list of all the patches available for a game. A patch is defined as the combination of a start version, end version, and distribution. You can mark the start version as "any" in which case it is assumed that if a more specific patch is not found, the "any" patch will be able to patch any version to the given end version. Distribution can also be selected as a specific distribution, or "any" if the patch can be used for any distribution.

The Patching Backend tries to locate the most-specific patch possible by trying to match the start version, end version (being the current version we're trying to patch to) and distribution as reported from the game. If an exact match is not found, it tries the more general cases (any start version, any distribution) to see if it can find a patch that will get the game to the current version. If it does not find a patch, it returns to the game as if the user has the current version. If it does find a patch, it returns the name of the new version, as well as the information needed to download the patch to the Patching SDK.

At the end of this process, the Patching SDK calls the game callback and indicates whether a patch is available, and if so, where to download it from. This whole process typically takes less than 1 second.

Once you've determined whether a patch is available, you can allow the user to choose to download and install it. The Patch Information Table can contain two different things that allow you to determine where to download a patch from. The first is just an HTTP URL to the patch executable. You can use the HTTP SDK or the external FPUUpdate utility (described below) to download and execute the patch. The other item the Patch Information Table can contain is a FilePlanet FileID number. This number can be used with FilePlanet and a web browser, allowing a user to download the file themselves.

Creating Patches

The Patching SDK does not provide any direct functionality for creating patches. We've found that developers and publishers typically already have their own systems for creating patches, or have already licensed a 3rd party patching tool.

Two products that we've seen successfully used are [RTPatch](#) and [Wise InstallMaster](#), but there are many others on the market.

Generally it's best if you consider your patching software and strategy before releasing your product. Some CD Copy-protection schemes can make patching difficult, and some products work better with some games than others, so we suggest testing any patching product with you game before it goes gold.

If you use the FPUdate (described below) to download and install the patch, the patch must be in a self-contained, self-installing EXE form. Most patching products are capable of creating patches in this form. If you manage the download / installation of the patch yourself, you can download the patch in whatever format you want.

One question that is often raised about our patching system is the ability to handle multi-part patches. That is, the ability to have one patch that goes from 1.0 to 1.5, another patch that goes from 1.5 to 1.7, a patch that goes from 1.7 to 2.0, and have the Patching SDK download all three files and run them in-order for version 1.0 clients. We decided not to directly support multi-part patching for a variety of reasons (mainly having to do with ambiguities in determining patching paths) and instead suggest that developers create and test full patches for each version they want to patch (e.g. 1.0 to 2.0, 1.5 to 2.0, and 1.7 to 2.0) or, create a small patch for the most recent version to the new version (e.g. 1.7 to 2.0), and then a larger patch that can patch any previous version to the new version (e.g. 1.x to 2.0). In our experience these two methods lead to better results compared to trying to install multiple generational patches. Even safer is creating a single patch that can be applied to any existing version to bring it to the new version (e.g. x.x to 2.0) but this often leads to larger patches.

For developers that still require mutli-part patches, we do have a solution available that works with the current Patching SDK. Contact [developer support](#) for more information.

File Manifest

The following files should be included with this package. If any of the files are missing, please [contact devsupport@gamespy.com](mailto:devsupport@gamespy.com).

File	Description
<i>ptMain.c</i>	Patching / Usage Analysis SDK code
<i>pt.h</i>	Patching / Usage Analysis SDK header file
<i>pt.dsp</i> Usage Analysis SDK	Example and test code for the Patching and Usage Analysis SDK
<i>pt.dsw</i>	Devstudio Workspace for API / sample code
<i>pttestc.c</i>	Example code
<i>pttestc.dsp</i>	Example code project

The HTTP SDK and GameSpy Common Code are also required for the Patching and Usage Analysis SDK.

Implementation

The following is a quick rundown of the various functions in the Patching SDK. The *pt.h* file contains additional documentation for each function.

ptCheckForPatch

This function is used to check if a patch is available for a certain version of a product. The three things that are used to uniquely identify a version of a product are passed in: product ID, version unique ID, and distribution ID. The SDK will then check for a new patch and call the callback. If the blocking parameter is **PTTrue**, then this function won't return until it finishes checking for a patch, or there is an error. If there is any sort of error initiating the check, then this function will return **PTFalse**, and the callback will not be called. For more info on the callback, see the description of **ptPatchCallback** below.

```
PTBool ptCheckForPatch
(
    int productID,
    const char * versionUniqueID,
    int distributionID,
    ptPatchCallback callback,
    PTBool blocking,
    void * param
);
```

productid

The product ID of the application for which to check.

versionUniqueID

The string that uniquely identifies this version. Max 30 characters.

distributionID

The distribution ID for this distribution of the application. Can be 0.

callback

This gets called with information about a possible patch.

blocking

If `PTTrue`, the function won't return until the callback has been called.

`param`

This is optional user-data that will be passed into the callback.

ptPatchCallback

This callback gets called as a result of the `ptCheckForPatch` function being called. See above for more info.

```
typedef void (* ptPatchCallback)
(
    PTBool available,
    PTBool mandatory,
    const char * versionName,
    int fileID,
    const char * downloadURL,
    void * param
);
```

`available`

`PTTrue` if a newer version is available. `PTFalse`, ignore the other parameters.

`mandatory`

If `PTTrue`, this patch has been marked as mandatory.

`versionName`

A user-readable display name for the new version.

`fileID`

A FilePlanet file ID for the patch. Can be 0.

Used to form a FilePlanet URL so the user can download the file.

`param`

This is optional user-data that was passed to `ptCheckForPatch`.

Using The Web Admin Interface

The web administration interface is located at <http://motd.gamespy.com/admin/patching/login.html> . It is used for administering the backend of the Patching SDK. The login system uses GameSpy ID for authentication. If you have not already done so, [create a GameSpy ID account](#) and send the e-mail address you used to devsupport@gamespy.com. You account will be given access to the admin page for your product, and you will be sent the productid number to use.

Login Screen

At the login screen, enter the login name, password, and productid you have been issued for your product.

Once your login has been verified, you will be sent to a page where you can access the various lists needed to administer the system.

Known Version List

The Known Version List contains a list of all of the versions that have been reported for your product. You can select a version and push the Edit button to edit it.

The two fields you can edit for each version are the name and the internal flag. The Version Name is used throughout the web interface to identify the version, and is sent to the Patching SDK by the Patching Backend when a client is notified of a new version. It should generally be a user-displayable string.

The internal flag is used to mark a version that should not be auto-updated (even if it doesn't match the current version). Typically this is used to flag internal or pre-release versions so that users testing them don't get update notifications to the current public version.

Versions are added to the Known Version List the first time a [versionUniqueID](#) is checked via the Patching SDK. The version list for your product will initially be empty - the first time you run a check with the

Patching SDK, whatever versionUniqueID you use will create a new entry in the Known Version List, which you can then edit via the web interface. Because entries are added automatically, you never need to worry about a mismatch between what is being reported via the Patching SDK and the versions listed on the web page.

Current Version List

The Current Version List is where you set which version is the most current for your product. If you are only using a single distribution, you should only have a single entry (with a distribution of "Normal" or "Any" - both will work). If you have multiple distributions that all have the same current version, you can simply set the distribution to "Any". If you need separate current versions for different distributions, you can add multiple entries.

Existing entries can be updated or deleted. You can simply update your entry when a new upgrade is available for a particular distribution.

Patch Information List

The Patch Information List contains the list of patches that are available for your product. You can add a new patch by specifying a start version, end version, and distribution. If the patch can be applied to any version to bring it to the end version, just select "Any" under start version. If the patch can be applied to any distribution (or you only have 1 distribution) select "Any" for the distribution - otherwise select the distribution the patch is appropriate for. You must add at least one of the two download location methods: Either a full URL or a FilePlanet FileID. You can enter data for both if you have both a FilePlanet Mirror and a separate web server mirror.

Existing patches can be updated with new locations or deleted. Generally you do not need to delete old patches, even if they aren't going to be used any more (because the version they patch to is no longer the current version), however you can delete patches if desired.

Patching and Usage Analysis SDK Functions

[ptCheckForPatch](#)

Determine whether a patch is available for the current version and particular distribution of a product.

[ptCheckForPatchAndTrackUsage](#)

Does the same thing as both [ptCheckForPatch](#) and [ptTrackUsage](#), in one call.

[ptTrackUsage](#)

Track usage of a product, based on version and distribution.

ptCheckForPatch

Determine whether a patch is available for the current version and particular distribution of a product.

```
PTBool ptCheckForPatch(  
    int productID,  
    const gsi_char * versionUniqueID,  
    int distributionID,  
    ptPatchCallback callback,  
    PTBool blocking,  
    void * instance );
```

Routine	Required Header	Distribution
ptCheckForPatch	<pt.h>	SDKZIP

Return Value

PTTrue is return if a query was sent. PTFalse means the operation was aborted.

Parameters

productID

[in] Numeric ID assigned by GameSpy. This is NOT the game ID.

versionUniqueID

[in] Developer specified string to indentify the current version.
Typically "1.0" form.

distributionID

[in] Optional indentifier for distribution. This is usually 0.

callback

[in] Function to be called when the operation completes.

blocking

[in] When set to PTTrue, this function will not return until the operation has completed.

instance

[in] Pointer to user data. This is optional and will be passed unmodified to the callback function.

Remarks

The **ptCheckForPatch** function sends a query to determine if a patch is available for the current game version and distribution. If this function does not return `PTFalse`, then the callback will be called with information on a possible patch to a newer version.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
ptCheckForPatch	ptCheckForPatchA	ptCheckForPatchW

ptCheckForPatchW and **ptCheckForPatchA** are UNICODE and ANSI mapped versions of **ptCheckForPatch**. The arguments of **ptCheckForPatchA** are ANSI strings; those of **ptCheckForPatchW** are wide-character strings.

Section Reference: [Gamespy Patching and Usage Analysis SDK](#)

ptCheckForPatchAndTrackUsage

Does the same thing as both ptCheckForPatch and ptTrackUsage, in one call.

```
PTBool ptCheckForPatchAndTrackUsage(  
    int userID,  
    int productID,  
    const gsi_char * versionUniqueID,  
    int distributionID,  
    ptPatchCallback callback,  
    PTBool blocking,  
    void * param );
```

Routine	Required Header	Distribution
ptCheckForPatchAndTrackUsage	<pt.h>	SDKZIP

Return Value

Parameters

userID

[in] Numeric ID assigned by GameSpy. This is NOT the game ID.

productID

[in] Developer specified string to indentify the current version.
Typically "1.0" form.

versionUniqueID

[in] Developer specified string to indentify the current version.
Typically "1.0" form.

distributionID

[in] Optional indentifier for distribution. This is usually 0.

callback

[in] Function to be called when the operation completes.

blocking

[in] When set to PTrue, this function will not return until the operation has completed.

param

[in] Pointer to user data. This is optional and will be passed unmodified to the callback function.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GS
ptCheckForPatchAndTrackUsage	ptCheckForPatchAndTrackUsageA	ptC

ptCheckForPatchAndTrackUsageW and **ptCheckForPatchAndTrackUsageA** are UNICODE and ANSI mapped versions of **ptCheckForPatchAndTrackUsage**. The arguments of **ptCheckForPatchAndTrackUsageA** are ANSI strings; those of **ptCheckForPatchAndTrackUsageW** are wide-character strings.

Section Reference: [Gamespy Patching and Usage Analysis SDK](#)

See Also: [ptCheckForPatch](#), [ptTrackUsage](#)

ptTrackUsage

Track usage of a product, based on version and distribution.

```
PTBool ptTrackUsage(  
    int userID,  
    int productID,  
    const gsi_char * versionUniqueID,  
    int distributionID,  
    PTBool blocking );
```

Routine	Required Header	Distribution
ptTrackUsage	<pt.h>	SDKZIP

Return Value

If PTFalse is returned, there was an error tracking usage.

Parameters

userID

[in] The GP userID of the user who is using the product. Can be 0.

productID

[in] The ID of this product.

versionUniqueID

[in] A string uniquely identifying this version.

distributionID

[in] The distribution ID for this version. Can be 0.

blocking

[in] When set to `PTTrue`, this function will not return until the operation has completed

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
ptTrackUsage	ptTrackUsageA	ptTrackUsageW

ptTrackUsageW and **ptTrackUsageA** are UNICODE and ANSI mapped versions of **ptTrackUsage**. The arguments of **ptTrackUsageA** are ANSI strings; those of **ptTrackUsageW** are wide-character strings.

Section Reference: [Gamespy Patching and Usage Analysis SDK](#)

Patching and Usage Analysis SDK Callbacks

[ptPatchCallback](#)

This callback gets called when a patch is being checked for with either `ptCheckForPatch` or `ptCheckForPatchAndTrackUsage`.

ptPatchCallback

This callback gets called when a patch is being checked for with either `ptCheckForPatch` or `ptCheckForPatchAndTrackUsage`.

```
typedef void (*ptPatchCallback)(  
    PTBool available,  
    PTBool mandatory,  
    const gsi_char * versionName,  
    int fileID,  
    const gsi_char * downloadURL,  
    void * param );
```

Routine	Required Header	Distribution
ptPatchCallback	<pt.h>	SDKZIP

Parameters

available

[in] PTTTrue if a newer version is available. If PTFalse, ignore the other parameters.

mandatory

[in] If PTTTrue, this patch has been marked as mandatory.

versionName

[in] A user-readable display name for the new version.

fileID

[in] A FilePlanet file ID for the patch. Can be 0. Used to form a FilePlanet URL

downloadURL

[in] If not an empty string, contains a URL to download the patch from.

param

[in] This is optional user-data that was passed to ptCheckForPatch.

Remarks

If a patch is available, and the fileID is not 0, then ptLookupFilePlanetInfo can be used to find download sites.

Section Reference: [Gamespy Patching and Usage Analysis SDK](#)

Peer SDK

Overview

The GameSpy Peer SDK is designed to provide an in-game, lobby interface for starting peer-to-peer games. The Peer SDK does this using several other GameSpy SDKs, including Chat (for chatting), Query and Reporting 2 (for server reporting), and ServerBrowsing (for server lists and server querying), but for the most part these interfaces are hidden from you, and you only need to work with the Peer SDK calls.

The Peer SDK only deals with data. You will be responsible for creating all the GUI elements that are required for the lobby system. Typically, this includes a scrolling list control (for the games list and chat participants list) a scrolling text window (for the chat window), buttons, and a text entry line (for the chat line). You may wish to create other controls to take advantage of the more advanced features of the SDK including player cross-pings and player status indicators.

Peer is designed for games that want to provide an in-game lobby system for setting up multiplayer games. Following is a description of how a game might typically use Peer. When the player first connects, they are placed in the main chat room for the game, called the "title room". Once in the title room, you can request a list of the current games being played. A list of games that are joinable will be returned, and it will be dynamically updated to add/remove/update games as changes occur in the list. Players can choose to either join one of the existing games, or create their own. When a player creates their own game and is waiting for others to join they are placed in a separate chat room called the "staging room". As other players join the staging room, ping measurements are exchanged, which can be used to determine the quality of the connections between the players (important for peer to peer games). Players can indicate their readiness and the host can choose to launch the game when ready. Once the host sends out the launch message, everyone in the staging room can then start playing the actual game.

Due to its flexibility, there are various ways to use Peer. For example, if a

game is joinable after it has been launched, the list of current games can include both games that are already running and those still in staging. The user could then have the option of joining a game in progress or joining a staging room.

Another option is to use group rooms to split the list of games into categories (by gametype, skill, region, etc.). In this case, when entering the title room, the user would get a list of group rooms instead of a list of games. They would see descriptions of the groups along with the number of players in each group. When the user selects a group, they would join that group's "group room". In here they can chat with others in the group room, and they would see a list of the games and/or staging rooms that are a part of this group. They could choose to either join one of the games, create their own room, or switch to another group.

A variation on this method would be to never join the title room - instead, the player would initially see just a list of group rooms they could join. This could help avoid people getting "stuck" chatting in the title room, without even seeing a list of games to play in. If Peer is just being used to report a game or get a list of servers, the application does not even need to connect to the chat server. After initializing and setting a title, just call `peerStartReporting()` to start reporting a server to the backend or `peerStartListingGames()` to start retrieving a server list.

File Manifest

The following files should be included with this package. If any of the files are missing, please [contact devsupport@gamespy.com](mailto:devsupport@gamespy.com).

File	Description
<i>peer.h</i>	GameSpy Peer header (all user functions are prototypes here)
<i>peerMain.c</i>	Entry point for all user Peer functions
<i>peerMain.h</i>	Common header for internal code
<i>peerCallbacks.c,h</i>	Code for queueing/calling callbacks
<i>peerSB.c,h</i>	Code for dealing with the Server Browsing SDK
<i>peerGlobalCallbacks.c,h</i>	Code for chat and QR2 callbacks
<i>peerKeys.c,h</i>	Code for handling global and room keys
<i>peerMangle.c,h</i>	Converts to and from room names
<i>peerOperations.c,h</i>	Code for running/maintaining operations
<i>peerPing.c,h</i>	Code for calculating player pings
<i>peerPlayers.c,h</i>	Keeps track of players
<i>peerRooms.c,h</i>	Keeps track of rooms
<i>peerHost.c,h</i>	Hosting rooms
<i>peerQR.c,h</i>	Reproting as a server and responding to queries
<i>peerAutoMatch.c,h</i>	AutoMatch functionality
<i>../nonport.c,h</i>	Platform-specific code
<i>../hastable.c,h</i>	Hastable
<i>../md5c.c,md5.h</i>	MD5 generation
<i>../darray.c,h</i>	Dynamic-Array
<i>../qr2/</i>	Server reporting
<i>../serverbrowsing/</i>	Server listing

<i>../pinger/</i>	UDP-pings
<i>../chat/</i>	Chat SDK
<i>/PeerLobby/</i> with a wizard-like interface	A sample application which uses the Peer SDK
<i>/PeerTest/</i> functionality	A test application used for testing specific Peer
<i>/PeerC/</i> C	A peer app written for the command-line in ANSI

Implementation

Initializing

Before doing anything else, Peer must be initialized with a call to `peerInitialize`:

```
PEER peerInitialize  
(  
    PEERCallbacks * callbacks  
);
```

`disconnected`

The chat connection has been disconnected by the server. You can attempt to reconnect with `peerConnect`, or shutdown with `peerShutdown`.

`roomMessage`

A chat message has arrived in one of the rooms the user is in.

`roomUTM`

An under-the-table message has arrived in a room the user is in.

`roomNameChanged`

The name of a room the user is in has changed.

`roomModeChanged`

The mode changed in a room the user is in.

`playerMessage`

A private chat message from another player has been received.

`playerUTM`

An under-the-table message has arrived from another player.

`readyChanged`

If the user is in a staging room, this will get called when one of the players changes his ready status (by default, all players are not ready).

`gameStarted`

If the user is in a staging room, this gets called when the host

launches the game. The host's IP is available as part of the callback, as well as a text string specified by the host.

[playerJoined](#)

A player has joined one of the rooms the local player has joined.

[playerLeft](#)

A player has left one of the rooms the local player has joined.

[kicked](#)

The local player has been kicked from a room.

[newPlayerList](#)

The entire playerlist has been updated, and should be checked with [peerEnumPlayers](#) if listing players.

[playerChangedNick](#)

When joining a room, this gets called for each player in the room when his IP and profile ID becomes available.

[playerFlagsChanged](#)

A player's room flags have changed.

[ping](#)

A new average ping time has been calculated for a player in a room being pinged. Which rooms get pinged is determined when the title is set with [peerSetTitle](#).

[crossPing](#)

A new average cross-ping time between two players is available. Which rooms get cross-pings is determined when the title is set with [peerSetTitle](#).

[globalKeyChanged](#)

A global watch key has changed or is newly available.

[roomKeyChanged](#)

A room watch key has changed or is newly available, or a broadcast key has changed.

[qrServerKey](#)

When reporting this is used to report values for server keys.

[qrPlayerKey](#)

When reporting this is used to report values for player keys.

qrTeamKey

When reporting this is used to report values for team keys.

qrKeyList

When reporting this is used to list the keys that will be reported.

qrCount

When reporting this is used to get the number of players and the number of teams.

qrAddError

This is used to notify the application of a server reporting error.

qrNatNegotiateCallback

This is used to pass nat-negotiate cookies to the server.

Peer is initialized until `peerShutdown` is called:

```
void peerShutdown
(
    PEER peer
);
```

Thinking

Once peer has been initialized, `peerThink` must be called frequently to allow Peer to do any necessary processing, including calling callbacks and processing pings. It should be called at least every 10 ms, in order to get accurate ping times. This is typically called in the program's main loop.

```
void peerThink
(
    PEER peer
);
```

Title

Setting a title tells peer which game it should be dealing with, and should be done after peer has been initialized but before it is connected. After

the title is set, peer can connect to the chat server, the title room can be joined, staging rooms can be created, and games and staging rooms can be joined. The title can later be changed without disconnecting. For information on getting your secret key, [contact developer support](#)

```
PEERBool peerSetTitle  
(  
    PEER peer,  
    const char * title,  
    const char * qrSecretKey,  
    const char * sbName,  
    const char * sbSecretKey,  
    int sbGameVersion,  
    int sbMaxUpdates,  
    PEERBool natNegotiate,  
    PEERBool pingRooms[NumRooms],  
    PEERBool crossPingRooms[NumRooms]  
);
```

peer

This is the peer object returned by [peerInitialize](#).

title

The title for the game. This controls what serverlist hosted games show up in and what games to show in a serverlist.

qrSecretKey

This is the secret key used by the QR2 SDK (which is used by the Peer SDK).

sbName, sbSecretKey

This is the server-browsing name and secret key used by the Peer SDK (which uses the ServerBrowsing SDK).

Used to form a FilePlanet URL so the user can download the file.

sbGameVersion

This is a number that uniquely identifies this version of the game.

sbMaxUpdates

This is the maximum number of servers to update at a time. This

should be 10-15 for modem users and 20-30 for high-bandwidth users.

`natNegotiate`

This should be set to `PEERTrue` if the game supports GameSpy's nat-negotiation technology (or a 3rd party solution).

`pingRooms`

Each element in this array should be set to `PEERTrue` to do pings in that room, or `PEERFalse` not to do pings in that room. For example:

```
pingRooms[TitleRoom] = PEERTrue;
pingRooms[GroupRoom] = PEERFalse;
pingRooms[StagingRoom] = PEERTrue;
```

`crossPingRooms`

Each element in this array should be set to `PEERTrue` to do cross-pings in that room, or `PEERFalse` not to do cross-pings in that room. For example:

```
crossPingRooms[TitleRoom] = PEERFalse;
crossPingRooms[GroupRoom] = PEERFalse;
crossPingRooms[StagingRoom] = PEERTrue;
```

peerClearTitle

`peerClearTitle` can be used to reset to no title. This can be useful during a game for freeing up resources and bandwidth, while not disconnecting totally from chat. However, if the game was launched from a staging room, there will be no way to return to the staging room after the game if `peerClearTitle` is called.

```
void peerClearTitle
(
```

```
); PEER peer
```

Connecting

Once Peer is initialized and a title is set, we can connect to the chat server. This is normally done with [peerConnect](#). However if the program needs to authenticate the user's login information with the chat server, [peerConnectLogin](#) or [peerConnectPreAuth](#) should be used. See the Logging In section for further details.

```
void peerConnect  
(  
    PEER peer,  
    const char * nick,  
    int profileID,  
    peerNickErrorCallback nickErrorCallback,  
    peerConnectCallback connectCallback,  
    void * param,  
    PEERBool blocking  
);
```

[peer](#)

This is the peer object returned by [peerInitialize](#).

[nick](#)

This is the nick with which to connect to the chat server. See below for chat nickname restrictions.

[profileID](#)

This is the local user's GP (GameSpy Presence and Messaging) profile ID. If the user doesn't have a GP account, or the profileID is not used by the program, this can be set to 0 and ignored. You can get another player's profileID with [peerGetPlayerProfileID](#) or [peerGetPlayerInfoNowait](#).

[nickErrorCallback](#)

If there was some sort of error with the nickname during the connection process, this callback is called. After a program

receives this notice, it can either try to continue the connect with a new nickname by calling `peerRetryWithNick` with the new nick, or it can stop the connection attempt by calling `peerRetryWithNick` with a `NULL` nick.

If the connection attempt is stopped, then the `connectCallback` will be called with a failure. If there is another nick error, `nickErrorCallback` will be called again. `PeerRetryWithNick` does not need to be called immediately - the program can prompt the user to try with a new nickname, then call `peerRetryWithNick` after the user selects a new one.

`connectCallback`

This gets called when the connection attempt completes.

`param`

User-data passed to both callbacks.

`blocking`

If `PEERTrue`, then the call won't return until the attempt has completed.

If successful, Peer will stay connected to chat until either `peerDisconnect` or `peerShutdown` is called:

```
void peerDisconnect  
(  
    PEER peer  
);
```

Peer uses the Chat SDK for all of its chat functionality. The `CHAT` object that Peer uses can also be used directly by the program to, for example, join a separate chat channel.

```
CHAT peerGetChat  
(  
    PEER peer  
);
```

Rooms

There are three types of rooms used by Peer: title rooms, group rooms, and staging rooms. The application uses these rooms to setup the path that the user takes between initially connecting and actually getting into a game. Each room type is optional, allowing for a variety of possible setups. Here is the first of two typical paths:

1. The user connects and is put into a title room, where he can chat with other users who are looking for a game to join. At this point the user can get a list of all joinable games, or create his own game.
2. The user joins/creates a game. If the game is joined and is already running, the user is launched directly into the game. Otherwise, the user is put into that game's staging room, where he can talk with other users getting ready to play the game.
3. When the user is prepared to play, he hits his "ready" button. After the host sees that everyone is ready, he hits his launch button, and everyone in the staging room gets launched directly into the game.

Another common path is similar to the above choice, but with the addition of group rooms between the title room and staging rooms (Note: to use group rooms, you must [contact developer support](#) to set them up):

1. The user connects and is put into a title room, where he can chat with other users who are looking for a game to join. At this point the user can see a list of group rooms, possibly sorted by skill level, location, or gametype.
2. The user picks a group and joins it. Now he can talk with other users that have chosen that group. The user can also get a list of all joinable games within the group, or create his own game within the group.
3. The user joins/creates a game. If the game is joined and is already running, the user is launched directly into the game. Otherwise, the user is put into that game's staging room, where he can talk with other users getting ready to play the game.
4. When the user is prepared to play, he hits his "ready" button. After the host sees that everyone is ready, he hits his launch button, and everyone in the staging room gets launched directly into the game.

Again, each room type is optional, so it is very easy to come up with a path that fits the needs of a particular game. For example, either of the above paths could be modified to skip the title room. Just don't join the title room and start off by showing a list of joinable games (or group rooms).

Title Rooms

There is one title room for each game. This is the main lobby where people can meet and chat while they look for a game to join. To join the title room, use `peerJoinTitleRoom`:

```
void peerJoinTitleRoom
(
    PEER peer,
    const char password[PEER_PASSWORD_LEN],
    peerJoinRoomCallback callback,
    void * param,
    PEERBool blocking
);
```

`peer`

This is the peer object returned by `peerIntialize`.

`password`

An optional password for the room, usually `NULL`.

`callback`

Gets called when the join completes or fails.

`param`

User-data passed to the callback.

`blocking`

If `PEERTrue`, then the call won't return until the attempt has completed.

To leave the title room, use `peerLeaveRoom` with the `roomType` set to `TitleRoom`.

Group Rooms

For certain applications it may be desirable to split up games (either in staging or already playing) into various groups. This can be done for several reasons, including categorizing servers by region ("Europe", "Asia", "North America", etc.), or to group players by skill level ("Newbie", "Intermediate", "Expert"). Peer allows this to be done by providing group rooms. When a user enters a group room, they will be able to chat with other players in that room, get a list of games in that group, and start a game in that group. To get a list of group rooms, use `peerListGroupRooms`. The `peerListGroupRoomsCallback` will be called once for each group room, then once again with a `groupID` of 0 to signal that there are no more groups. If you want to use group rooms in a game, contact devsupport@gamespy.com to get them set up.

```
void peerListGroupRooms
(
    PEER peer,
    const char * fields,
    PeerListingGroupRoomsCallback callback,
    void * param,
    PEERBool blocking
);
```

`peer`

This is the peer object returned by `peerIntialize`.

`fields`

This is an optional backslash-delimited list of extra keys/values to get for each group room.

`callback`

Gets called once for each group room, and once more to signal the end of the list.

`param`

User-data passed to the callback.

`blocking`

If `PEERTrue`, then the call won't return until the attempt has

completed.

```
typedef void (* peerListGroupRoomsCallback)
(
    PEER peer,
    PEERBool success,
    int groupID,
    SBServer server
    const char * name,
    int numWaiting,
    int maxWaiting,
    int numGames,
    int numPlaying,
    void * param
);
```

peer

This is the peer object returned by `peerIntialize`.

success

This will be `PEERFalse` if there is an error listing groups. If there is an error, there will be no more calls to the callback.

groupID

This is unique identifier for the group, and it is used when joining a group room. If there is no error, and this is 0, it is signaling that there are no more groups to be listed. If it is 0, then name will be `NULL`, and `numWaiting`, `maxWaiting`, `numGames`, and `numPlaying` will all be 0.

server

This server object may contain extra key/value information for this group.

name

The name of the group.

numWaiting

The number of players in the group room.

maxWaiting

The maximum number of players allowed in the group room.

`numGames`

The number of games currently in this group, either in staging or already running.

`numPlaying`

The total number of players in all of this group's games.

`param`

User-data passed to `peerListGroupRooms`.

To join a group room, use `peerJoinGroupRoom`. Once the room has been joined, the listing of games will be filtered so that only games that are in the same group are listed. If a game listing is in progress when a group room is joined (or left), the listing will be cleared and started over (the callback will be called with `msg==PEER_CLEAR`).

```
void peerJoinGroupRoom ( PEER peer, int groupID,  
peerJoinRoomCallback callback, void * param, PEERBool blocking );
```

`peer`

This is the peer object returned by `peerInitialize`

`groupID`

The ID of the group to join (as passed to the `peerListGroupRoomsCallback`).

`callback`

Gets called when the join completes or fails.

`param`

User-data passed to the callback.

`blocking`

If `PEERTrue` then the call won't return until the attempt has completed.

To leave a group room, use `peerLeaveRoom` with the `roomType` set to `GroupRoom`.

Staging Rooms

Staging rooms are chat rooms where players can join up and chat before launching into a game. To create a staging room, use `peerCreateStagingRoom`. Once a staging room has been created, the six QR callbacks that were specified as part of `peerInitialize` will be called periodically to get information on the server. This will last until the host leaves the staging room (or, if the host has started a game and then left the staging room, until the game stops).

If the user is in a group room when the staging room is created, the staging room will be reported as part of that group. This association will stick even if the player then leaves the group room.

```
void peerCreateStagingRoom
(
    PEER peer,
    const char * name,
    int maxPlayers,
    const char password[PEER_PASSWORD_LEN],
    peerJoinRoomCallback callback,
    void * param,
    PEERBool blocking
);
```

`peer`

This is the peer object returned by `peerInitialize`

`name`

The name to give the room.

`maxPlayers`

The maximum number of players to allow in the staging room.

`password`

An optional password for the staging room.

`callback`

Gets called when the create completes or fails.

`param`

User-data passed to the callback.

blocking

If `PEERTrue` then the call won't return until the attempt has completed.

`peerStartListingGames` is a way for the program to get a dynamic list of all games for the current title. The callback is repeatedly called to let the program know what to do to make its game list current. This continues until `peerStopGames` is called. If the user is in a group room, only the games for that group will be listed. If the user is not in a group room, games that are not part of any group room will be listed.

After `peerStartListingGames()` completes its initial list of all available game servers, it goes into automatic update mode, where game updates are propagated as they are reported by the games themselves to the master. The list of keys you receive during this update phase is determined by a specific list of push keys that have been defined for your title. Push keys come directly from the master server and avoid any NAT/firewall problems the host may be having.

The set of push keys differs from the initial list requested via the fields array, as you may care less about certain keys during updates (the hostname of a game is not likely to change e.g.). By default, the BASIC keys that are pushed from the master are: **hostname**, **mapname**, **gametype**, **numplayers**, **maxplayers**, **country**, **gamemode**, **password** and **gamever**. You can contact [developer support](#) and request a modified list for your title. (Please make sure to include in your e-mail the ascii names of the keys to be added, ie. "mapname", "gametype", etc.)

The maximum number of push keys that can be sent out (including the default keys) is 50. The string listing all the keys (including the delimiting backslashes) can be up to 256 characters. The total list of name value pairs returned (including backslashes) can be up to 1024 characters.

```
void peerStartListingGames
(
    PEER peer,
    const unsigned char * fields,
    int numFields,
```

```
    const char * filter,  
    peerListingGamesCallback callback,  
    void * param  
);
```

peer

This is the peer object returned by `peerInitialize`

fields

An array of registered QR2 keys to request from servers.

NumFields

The number of keys in the array.

filter

This is a SQL-style filter that is applied to the initial listing of servers.

callback

Gets called each time there is a change in the game list.

param

User-data passed to the callback.

```
typedef void (* peerListingGamesCallback)  
(  
    PEER peer,  
    PEERBool success,  
    const char * name,  
    SBServer server,  
    PEERBool staging,  
    int msg,  
    int progress,  
    void * param  
);
```

peer

This is the peer object returned by `peerInitialize`

success

This will be `PEERFalse` if there is an error listing games. The

listing stops as soon as that happens.

name

The name of the game.

server

The ServerBrowsing [SBerver](#) object for this game. This can be used to get various information about the game, including ping, number of players, player names and pings, etc. See the ServerBrowsing SDK documentation and the bottom of *serverbrowsing\sb_serverbrowsing.h* for further information. It is also used as a way of uniquely identifying a game. The server object for a game is the same object from the time it gets added with [PEER_ADD](#), through any [PEER_UPDATE](#)'s, until its removed with [PEER_REMOVE](#).

The server object should be stored for each game listed as a way of identifying it when a [PEER_UPDATE](#) or [PEER_REMOVE](#) is sent for it. This parameter is [NULL](#) if the msg is [PEER_CLEAR](#) or [PEER_COMPLETE](#).

staging

If this is [PEERTrue](#), then this game has not been launched yet, and is still in the staging room. That means that this game can be joined with [peerJoinStagingRoom](#). If this is [PEERFalse](#), this game is already running. In this case, the application can just join the game whenever it wants by getting any necessary info from the server object (such as address with [ServerGetAddress](#)).

progress

When first starting to list games, an initial list of current games is received, then updated as new game are started and old games are updated or removed. While the initial listing is happening, this lets the program know what percentage of the initial list has been added so far. It will start at 0 with the [PEER_CLEAR](#) message, then rise up to 100 with the [PEER_COMPLETE](#) message. When it reaches 100, it will stay there until the listing is stopped.

param

User-data passed to [peerStartListingGames](#).

Possible `msg` types are:

PEER_CLEAR

Clear the list. This has the same effect as if a `PEER_REMOVE` were sent for every game listed. One of these is sent initially when listing starts, and it is also sent if a group room is joined or left while games are being listed. The server object is `NULL` for this type.

PEER_ADD

This is a new game. Add it to the list.

PEER_UPDATE

This game is already on the list, and its been updated. To match this game up to the one in your internal list, use the server object. If the program is only listing server names this can be ignored.

PEER_REMOVE

Remove this game from the list. Use the server object to match up the game to the one in your internal list. The server object is valid during this call, but will become invalid immediately after the call, and so should NOT be used after returning from the callback.

PEER_COMPLETE

The listing of current servers is complete. The application will now get dynamic updates as servers get started, get updated, or get shutdown. The server object is `NULL` for this type.

To join a staging room, use `peerJoinStagingRoom`. NOTE: These should only be used for games listed with staging set to `PEERTrue`. If this is set to `PEERFalse`, the game is already running, and the staging room cannot be joined.

```
void peerJoinStagingRoom
(
    PEER peer,
    GServer server,
    const char password[PEER_PASSWORD_LEN],
    peerJoinRoomCallback callback,
    void * param,
    PEERBool blocking
```

```
);
```

peer

This is the peer object returned by [peerInitialize](#)

server

The server object received when listing games.

password

The password for this room. Ignored if the room has no password.

callback

Gets called each time there is a change in the game list.

param

User-data passed to the callback.

blocking

If [PEERTrue](#) then the call won't return until the attempt has completed.

To leave a staging room, use [peerLeaveRoom](#) with the [roomType](#) set to [StagingRoom](#).

Messaging

To send a message to a room the user is in, use [peerMessageRoom](#):

```
void peerMessageRoom
(
    PEER peer,
    RoomType roomType,
    const char * message,
    MessageType messageType
);
```

peer

This is the peer object returned by [peerInitialize](#)

roomType

The room to send the message to: [TitleRoom](#), [GroupRoom](#), or

StagingRoom.

`message`

The message to send.

`messageType`

The type of message to send: NormalMessage, ActionMessage, NoticeMessage.

Players

Listing

To enumerate through all of the players in a room, use `peerEnumPlayers`. This is done using a local list maintained by Peer, and so it will do the enumerating before returning.

```
void peerEnumPlayers
(
    PEER peer,
    RoomType roomType,
    peerEnumPlayersCallback callback,
    void * param
);
```

`peer`

This is the peer object returned by `peerInitialize`

`roomType`

The room for which to list the players.

`callback`

Gets called once for each player in the room, and then once at the end of the listing (or once if there's an error).

`param`

User-data passed to the callback.

This callback gets called for each player in the room:

```
typedef void (* peerEnumPlayersCallback)
(
    PEER peer,
    PEERBool success,
    RoomType roomType,
    int index,
    const char * nick,
    PEERBool host,
    void * param
);
```

peer

This is the peer object returned by `peerInitialize`

success

If this is `PEERFalse`, there has been an error.

roomType

The room for which to list the players.

index

The index of the player, from 0 to the one less than the total number of players (N - 1). Or, if this is -1, that means the enumerating has completed.

nick

The nick of this player.

host

`PEERTrue` if tis player is the host of the room (this is equivalent to having operator privileges in a chat channel).

param

User-data passed to the `peerEnumPlayers`.

Messaging

To send a private message to another player, use `peerMessagePlayer`.

```
void peerMessagePlayer
(
  PEER peer,
  const char * nick,
  const char * message,
  MessageType messageType
);
```

peer

This is the peer object returned by [peerInitialize](#)

nick

The nick of the player to send them message to. See Nickname restrictions for valid chat nicks.

message

The message to send.

messageType

The type of message to send: NormalMessage, ActionMessage, NoticeMessage.

Flags

Every player has a set of flags associated with them in each room they are in. Flags are reported in the [peerFlagsChangedCallback](#), and can also be checked at any time with [peerGetPlayerFlags](#):

```
PEERBool peerGetPlayerFlags
(
    PEER peer,
    const char * nick,
    RoomType roomType,
    int * flags
);
```

peer

This is the peer object returned by [peerInitialize](#)

nick

The nick of the player to get flags for. See Nickname restrictions for valid chat nicks.

roomType

The room to get the flags for.

flags

The address at which to store the flags.

The flags can be any combination of the following bit defines:

PEER_FLAG_STAGING

in a staging room

PEER_FLAG_READY

ready in a staging room

PEER_FLAG_PLAYING

playing a game

PEER_FLAG_AWAY

set as away

PEER_FLAG_HOST

host of the room

PEER_FLAG_OP

has operator privileges in the room

PEER_FLAG_VOICE

has voice (+v) in the room

Launching

Once a staging room has been created, and usually after more players have joined the room, the host can launch the game itself. When the host chooses to launch the game with `peerStartGame`, every player in the staging room will get the `peerGameStartedCallback`. The game should be launched immediately after the call for the host, and as soon as the callback is called for other players. All players must call `peerStopGame` when either the game ends or they leave.

Ready

Each player in a staging room has a ready state, which is on or off. It is initially off when a staging room is joined. To get a player's ready state use `peerGetReady`. Whenever a player's ready state changes, the `peerReadyChangedCallback` will be called. `peerAreAllReady` is a utility function that checks if all the players in the staging room are ready. It can be used by the program to determine if the host can launch the game or not. To set your ready state, use `peerSetReady`.

Starting

When the host is ready to start the game, the program should call `peerStartGame`. It will cause everyone in the staging room (except for the host) to have their `peerGameStartedCallback` called. The host can leave the staging room once the game has started. However, the host won't be able to get back into the staging room, a new one would need to be created.

To maintain compatibility with GameSpy Arcade, the message string should be of the form "<dotted-IP>[:<port>]". Dotted-IP is the IP of the server in string form. This is optionally followed by the port the game is being hosted on. If the port is not used, the default port for the game will be assumed. To get the local IP, call `peerGetLocalIP()`, which returns the IP (in network byte order).

```
void peerStartGame
(
    PEER peer,
    const char * message,
    int reportingOptions
);
```

peer

This is the peer object returned by `peerInitialize`

message

This is a text string that all the other players will get as part of the

[peerGameStartedCallback](#). See the above paragraph for an explanation of the message.

ReportingOptions

This determines if Peer should continue reporting the game, or if it should stop and let the program take over. For games that use Peer internally, it is recommended that they set the [PEER_KEEP_REPORTING](#) flag and let Peer handle server reporting. For games that are launched externally, from GameSpy Arcade, for example, it will be necessary to stop reporting with [PEER_STOP_REPORTING](#) and let the external process take over the reporting. If Peer continues to report, [PEER_REPORT_INFO](#) and [PEER_REPORT_PLAYERS](#) can be used to control what information Peer reports.

Stopping

After the host has started a game, it has to let Peer know when the game has stopped. This is done with a call to [peerStopGame](#). This lets Peer either stop reporting the game (if the host has left the staging room), or to return to reporting it as a staging room.

This call should also be used by clients after a game they were playing in has finished. This allows peer to correctly report if this player is in game or not.

```
void peerStopGame
(
    PEER peer
);
```

Logging In

In the Connecting section above, [peerConnect](#) is shown as the function to use when connecting to the chat server. However there are a couple of other functions that can be used to not only connect to the chat server, but to also login using account information. These two functions are

[peerConnectLogin](#) and [peerConnectPreAuth](#).

In total, there are five different options for connecting:

1. Anonymous Login ([peerConnect](#))

This is the function to use if you want to connect to Peer without logging in or authenticating any user information. You will be able to use Peer normally, however you won't have uniquenicks, and there will be no way to verify that a given user is really who they say they are. The player's chat nick will be the nick passed to [peerConnect](#). See Nickname restrictions for valid chat nicks.

2. GameSpyID Login with no uniquenick ([peerConnectLogin](#) with a [namespaceID](#) of 0)

This is the method to use if you want to login to the GameSpyID system, but don't want to use uniquenicks. You'll use [peerConnectLogin](#) with the email, profilenick, and password for the account you are attempting to login under, and you'll set the [namespaceID](#) to 0. This is the "null" namespace, and is used to tell Peer that it should not set a namespace. The player's chat nick will be the profilenick passed to [peerConnectLogin](#). If the [profilenick](#) is an invalid chat nick, or is already in use on the server, the [nickErrorCallback](#) will be called. Note that the chat nickname rules apply. See Nickname restrictions for valid chat nicks.

3. GameSpyID Login with a uniquenick in the default namespace ([peerConnectLogin](#) with a [namespaceID](#) of 1)

This method is similar to the above method, however the [namespaceID](#) is set to 1, indicating the default GameSpy namespace. This is the same namespace that is used by GameSpy Arcade. The login information you pass to [peerConnectLogin](#) will be the profile's email, nick, and password. You do not need to pass the uniquenick for this method. The provided information will

uniquely identify a GameSpyID profile account. When logging in with this method, the chat nickname will be the profile's uniquenick with "-gs" appended. "-gs" is the namespace extension for the default GameSpy namespace. For example, if a user has the uniquenick Joe, his chat nick will be "Joe-gs". [peerTranslateNick](#) can be used to strip extensions off of nicks.

If the profile does not have a uniquenick associated with it in the GameSpy namespace, then the [nickErrorCallback](#) will be called with a type of [PEER_NO_UNIQUNICK](#). If there is a uniquenick, but it has expired, then the [nickErrorCallback](#) will be called with a type of [PEER_UNIQUNICK_EXPIRED](#). In either of these two cases, the application should use [peerRegisterUniqueNick](#) to register a uniquenick for the profile. If there is a problem registering the uniquenick, such as it being invalid or already in use, then the [nickErrorCallback](#) will be called again with a type of [PEER_INVALID_UNIQUNICK](#), and the [suggestedNicks](#) field will be filled in with suggestions. In this case [peerRegisterUniqueNick](#) should be called again, and continue to be called until a valid nick is registered. When this happens, the [connectCallback](#) will be called indicating a successful login.

[peerConnectLogin](#) should not be used with just a uniquenick and password when in the default namespace, or in any other namespace where uniquenicks can expire. This is because if a user's uniquenick has expired, and another user has since registered that uniquenick, then the user will no longer be able to login with just that uniquenick and password. Unique nicks have similar restrictions as chat nicks. See Nickname restrictions for valid unique nicks.

4. GameSpyID Login with a uniquenick in a custom namespace ([peerConnectLogin](#) with the custom [namespaceID](#))

You should only be using this method if you have been assigned a custom namespace. You can [contact devsupport@gamespy.com](mailto:contact_devsupport@gamespy.com) for information about getting a custom namespace.

If the namespace has expiring uniquenicks, then this method is almost identical to the above method, with the exception of using a custom [namespaceID](#) instead of 1 for the default namespace.

If the namespace does not have expiring uniquenicks, then the main difference is that [peerConnectLogin](#) can be used with just a uniquenick and password instead of the email, nick, and password used in the above method.

Another difference between this method and the above method is the namespace extension. The default GameSpy namespace has an extension of "-gs", while other namespaces have their own unique extensions. When you are assigned a custom namespace, you will be given the custom extension for use in your namespace. The chat nick for a player in this namespace will be his uniquenick with the namespace extension appended. For example, if a user has the uniquenick Joe in the GameSpy test namespace, his chat nick will be "Joe-gmt". [peerTranslateNick](#) can be used to strip extensions off of nicks. See Nickname restrictions for valid unique nicks.

5. Remote Authentication ([peerConnectPreAuth](#))

The remote authentication login method is used to login using information from a partner authentication system. You login using a token and a challenge, which are supplied by the partner authentication system. [Contact devsupport@gamespy.com](mailto:devsupport@gamespy.com) for further information on using this login method.

Nickname Restrictions

There are three different nicknames used in the Peer. A profile nick passed to peerConnectLogin is only restricted to all characters except the "\" character and a limit of 30 characters. The chat and unique nicks have more restrictions. The character limit for both nicks is 20. Chat and Unique nicks have the following restrictions:

- The first character cannot be one of the following characters: +, @, #, :
- Numeric characters are only allowed after the first character.
- All characters in the ASCII character range 34-126 are valid except for the backslash character (character 92, "\").

UNICODE Support

The GameSpy SDKs support an optional UNICODE interface for widestring applications. To use this interface, first define the symbol "[GSI_UNICODE](#)". Then, use widestrings wherever ANSI strings were previously called for. When in doubt, please refer to the header files for specific function declarations.

Although the GameSpy SDK interfaces support UNICODE parameters, some items may be stripped of their extra UNICODE information. These items include: nickname, email address, and URL strings. You may pass in widestring values, but they will first be converted to their ANSI counterparts before transmission.

Peer AutoMatch

Overview

The ability to automatically match players together, or AutoMatch, was added to the Peer SDK with version 2.01. The system is designed to be very flexible, allowing the application to use arbitrary values to rate possible matches, which helps to ensure that the best possible match is made.

To start, the application needs to get the local player's preferences, then start an AutoMatch attempt. The application is then responsible for rating potential matches and handling queries as to the local player's preferences. The local player will be placed in a staging room while waiting for a match and will be able to chat with the other players while waiting for a full match to be set up. The application does not need to present a chat interface or allow the user to chat. Once all the players in a match are in a room together, the application uses the [peerStartGame\(\)](#) function to start the actual game.

The two most important aspects of implementing AutoMatch are determining what choices the user will have and how potential matches will be rated. The choices/preferences available to the user will determine when values need to be reported in the [peerQR*Callback\(\)](#) functions, how to setup the filter passed to [peerStartAutoMatch\[WithSocket\]](#), and what values are available when rating potential matches. The method for rating potential matches will determine how Peer decides which match is the best and will dictate what values need to be reported in the [peerQR*Callback\(\)](#) functions.

Implementation

Starting An Automatch

There are only a few functions directly involved in starting and running an AutoMatch. The basics of getting connected to the chat backend are the same as with regular Peer matchmaking. This involves calling `peerInitialize()`, `peerSetTitle()`, and `peerConnect()`, and using `peerThink()` to allow Peer to do any needed processing. See the "GameSpy Peer SDK" document for further information, specifically the sections Initializing, Thinking, Title, and Connecting. Once connected to the backend, Peer is ready to do AutoMatching.

To start the AutoMatch attempt, call either `peerStartAutoMatch()` or `peerStartAutoMatchWithSocket()`. The `WithSocket` version of the function allows the application to provide a UDP socket that Peer will be used for reporting. Both functions take a `maxPlayers` parameter, which specifies the maximum number of people that should be in the final match. For example, if the local player wants to play a 3v3 match, `maxPlayers` should be 6. The match can be started before the `maxPlayers` is reached, for example if an exact number of players is not needed, but an upper limit must still be specified.

The filter is a SQL-type filter, just like the filter used in `peerStartListingGames()`. It is used to rule out matches that are not acceptable. Potential matches that do not pass the filter will not be passed to the rating callback. The `statusCallback` is called whenever the status of the match changes, until either `peerStopAutoMatch()` is called, or the `statusCallback` is called with a status of `PEERFailed` or `PEERComplete`. The `rateCallback` is used to rate possible matches.

Once an AutoMatch attempt has been started, Peer handles everything, calling the `rateCallback` and the `peerQR*Callback()` functions whenever they are needed. The application is then responsible for assigning ratings to servers in the `rateCallback`, responding to queries through the `peerQR*Callback()` functions, updating the UI based on the `statusCallback` or `peerGetAutoMatchStatus()`,

having the host start the game when reaching the `PEERReady` status, and having non-host players watch for the `peerGameStartedCallback` and/or the `PEERComplete` status. The application should also allow the user to cancel the `AutoMatch` attempt with `peerStopAutoMatch()`.

Rating A Potential Match

During an `AutoMatch` attempt `Peer` may ask the application, through the `peerAutoMatchRateCallback()` passed to `peerStartAutoMatch[WithSocket]()`, to rate a potential match. The application is responsible for assigning an integer rating value to the match, which is returned from the callback.

`Peer` uses the rating to determine if the match is acceptable and, if it is acceptable, how good of a match it is. If a value of 0 or less is returned from the callback, `Peer` will not attempt to join up to that match. If a value is 1 or greater than `Peer` may attempt to join the match. The higher the value returned, the better the match. If there are multiple acceptable matches, `Peer` will attempt to join them in order starting with the highest rated match, then the second highest rated match, etc.

The rating callback must take into account all of the local player's preferences compared to the settings/preferences for all of the players currently in the match, and come up with a single number representing how good the potential match is. Typically the callback will first check any "hard criteria", which are any settings that must match. For example if the player has selected that he only wants to play in a 2v2 match, the first line of the callback may check that the match is a 2v2 match:

```
if(SBServerGetIntValue(match, "maxplayers", 0) != 4)
    return 0;
```

A good method for comparing a list of "soft criteria" (such as preferences) is to assign each value a maximum weight, calculate the actual weight for each value by comparing the local value to the value reported by the server, then total all the weights and return that value as the rating. If a value is to be compared against the values of each player already in the

match, then the existing players' values can be averaged, then compared against.

A game could, for example, assign maximum weights of 100 to ratings differences and 50 to map preferences. In the callback the application averages the players' ratings and compares them to the local player's rating, determining that, because there is a fairly large difference, the ratings differences actual weight is 25. The application then compares the local player's map preferences to each player's preferences, assigns a weight to each player based on how close the preferences match, then averages all of those individual weights to determine the overall actual weight for map preferences.

Because their preferences are fairly close, the actual weight for map preferences is 40. Adding together the actual weights for ratings differences and map preferences gives a total of 65, which the application then returns from the callback as the rating for the match.

Automatic Hosting

During an AutoMatch attempt, if the local player ends up as host of a staging room (see above for how a user may end up in the `PEERwaiting` status), then the `peerQR*Callback()` functions that were registered with `peerInitialize()` will be called whenever the local player is queried for his preferences. The application uses these callbacks to report the local player's preferences/settings for the AutoMatch. Any information that other players may need to decide if the local player is a suitable match should be reported.

It is entirely up to the application to decide what information is needed and what information to report. Typically this would be information such as the local player's rank or rating, map preference, gametype preference, and/or the number of players to play with (`maxplayers`).

Because players use the reported information to decide if they want to join the local player's match, the local player must report both his own information and the information for any other players already in his staging room. This is because players that are looking for a match need to know if the match as a whole is suitable, and they may need to decide

that based on all the players already in the room. For example, if one of the available settings is a yes/no preference for each of the available maps, then a user looking for a match will want to compare his preferences against the preferences of all the players already in the match.

The host of the room can report each player's preferences as a player key. Those looking for a match can then determine a score for map preferences by comparing his preferences to each of the other player's preferences. For more information on how to report this information, see the "GameSpy Peer SDK Reference" document, the "GameSpy Query and Reporting 2 SDK" document, and the Peer samples.

Peer SDK Functions

peerAlwaysGetPlayerInfo	Tell the peer SDK to always retrieve IP and profile information for room members.
peerAreAllReady	Used to check if all players in the staging room are ready.
peerAuthenticateCDKey	Allows pre-chat cd key authentication via the chat server.
peerChangeNick	Change the chat nickname associated with the local client. This does not affect the account name.
peerClearTitle	Resets the peer SDK. <code>peerSetTitle</code> must be called before new operations are made.
peerConnect	Connect to the chat server.
peerConnectLogin	Connects a peer object to the backend chat server and then logs in using an account from the GameSpy Presence system. A title must be set with <code>peerSetTitle()</code> before this function is called.
peerConnectPreAuth	Connects a peer object to the backend chat server and then logs in

	using authentication information from a parter authentication system.
peerCreateStagingRoom	Creates a new staging room with the local player as the host.
peerCreateStagingRoomWithSocket	Creates a new staging room with the local player as the host.
peerDisconnect	Disconnect from the chat server. peerShutdown must still be called.
peerEnumPlayers	Enumerates through the players in a room.
peerFixNick	Repairs an illegal chat nickname. Removes illegal characters from a nickname as well as invalid character combinations.
peerGetAutoMatchStatus	Used when automatching to retrieve the current status.
peerGetChat	Returns the chat sdk object.
peerGetGlobalWatchKey	Returns the cached value of a players watch key.
peerGetGroupID	Returns the current group ID set from peerJoinGroupRoom or peerSetGroupID.

peerGetHostServer	Returns the SBServer object associated with the staging room host.
peerGetNick	Returns the chat nickname of the local client.
peerGetPlayerFlags	Returns the cached flag values for the current player. This is from the key "b_flags".
peerGetPlayerGlobalKeys	Query the server for a player's global key values.
peerGetPlayerInfo	Retrieve a local player's IP and profile ID.
peerGetPlayerInfoNoWait	Retrieve a player's IP and profile ID. Uses a cached copy.
peerGetPlayerPing	Returns the cached ping between the local player and the specified remote player.
peerGetPlayersCrossPing	Calculates the cross-ping between 2 players.
peerGetPrivateIP	Returns the local private IP address.
peerGetProfileID	

	Returns the profile ID of the local client. Only valid with peerConnectLogin or peerConnectPreAuth.
peerGetPublicIP	Returns the local public IP address.
peerGetReady	Get the ready state of the specified player.
peerGetRoomChannel	Returns the chat channel associated with the room type.
peerGetRoomGlobalKeys	Retrieves global keys for all players in the specified room. (Local client must be a room member.)
peerGetRoomKeys	Retrieves room key values for the room or a single player.
peerGetRoomName	Returns the channel's title. The local client must be a member of the room.
peerGetTitle	Gets the currently set title
peerGetUserID	Returns the local userID. Only valid with peerConnectLogin or peerConnectPreAuth.
peerInitialize	Initialize the peer SDK.

peerInRoom	Determines if the local client is in a room of the specified type.
peerIsAutoMatching	Returns PEERTrue if an AutoMatch is in progress.
peerIsConnected	Returns PEERTrue if connected to the chat server.
peerIsPlayerHost	Returns PEERTrue if specified player is a room host or operator.
peerIsPlaying	Returns PEERTrue if the local client is playing.
peerJoinGroupRoom	Join the specified group room.
peerJoinStagingRoom	Joins a specified game staging room. Allows players to get together and chat while setting up a game. Players can also see other players' pings and crosspings.
peerJoinStagingRoomByChannel	Join a staging room using the channel name.
peerJoinTitleRoom	Join the title room for the local client's game application.
peerKickPlayer	

	Kick a player from a room.
peerLeaveRoom	Remove the local client from a room.
peerListGroupRooms	List all the group rooms for the currently set title.
peerMessagePlayer	Send a message to the specified player.
peerMessageRoom	Send a message to the specified room
peerParseQuery	Pass a manually received server query to the peer SDK. Use with <code>peerStartReportingWithSocket</code> or <code>peerCreateStagingRoomWithSocket</code>
peerPingPlayer	Send a ping request to a remote player.
peerRegisterUniqueNick	Register a unique nick. Call in response to <code>peerNickErrorCallback</code> .
peerRetryWithNick	Use in response to a <code>nickErrorCallback</code> . This function allows the local client to retry the connection attempt with a different chat nickname.
peerSendNatNegotiateCookie	Send a nat negotiation cookie to a

	server.
peerSetAwayMode	Set the away mode, as it appears in chat.
peerSetGlobalKeys	Set global keys on the local player.
peerSetGlobalWatchKeys	Set the global watch keys for the specified room type.
peerSetGroupID	Manually set the group ID. Use with caution as this is normally set automatically.
peerSetPassword	Set the password on the chat room. Local client must be the host.
peerSetQuietMode	Sets the peer sdk to quiet mode or disables quiet mode. See remarks.
peerSetReady	Set the local clients ready state.
peerSetRoomKeys	Set room keys for a player or the room itself.
peerSetRoomName	Set the name of a room. Local client must be the host.
peerSetStagingRoomMaxPlayers	Update the maximum number of players for a staging room. Local

	client must be the host.
peerSetTitle	Set the game information to be used by the peer sdk.
peerSetTitleRoomChannel	Set the channel to be used as the TitleRoom. (Rarely used, SDK sets title room automatically.)
peerShutdown	Destructs the peer SDK.
peerStartAutoMatch	Start an automatch attempt.
peerStartAutoMatchWithSocket	Start an automatch attempt using an external managed socket.
peerStartGame	Called by the host to begin the game.
peerStartListingGames	Begin listing the currently running games and staging rooms.
peerStartPlaying	Flag the local player as "playing". Use this to manual set the player's flag in the event a non peer sdk game is started.
peerStartReporting	Begins server reporting, does not create a staging room.
peerStartReportingWithSocket	

	Begin server reporting using an externally managed socket.
peerStateChanged	Notify the backend of a server state change, such as the server becoming full.
peerStayInRoom	Allows SDK to remain in the title room after peerClearTitle. (Rarely used.)
peerStopAutoMatch	Stop an automatch attempt in progress.
peerStopGame	Called by the host when the game has ended.
peerStopListingGames	Stops a server list update in progress. Also stops listening for game state changed messages.
peerThink	Allow the Peer SDK to continue processing. Callbacks will be triggered during this call.
peerTranslateNick	Removes the namespace extension from a nickname. Use this when working with unique nicknames in a public chat room.
peerUpdateGame	Send an update query to the specified server.

[peerUpdateGameByMaster](#)

This function updates a server via the master server. Passing in true for fullUpdate will obtain the full keys for that server, otherwise it will only obtain the basic keys.

[peerUTMPlayer](#)

Send a UTM message to the specified client.

[peerUTMRoom](#)

Send a UTM message to each client in the room.

peerAlwaysGetPlayerInfo

Tell the peer SDK to always retrieve IP and profile information for room members.

```
void peerAlwaysGetPlayerInfo(  
    PEER peer,  
    PEERBool always );
```

Routine	Required Header	Distribution
peerAlwaysGetPlayerInfo	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object.

always

[in] Set to PEERTrue to have the SDK automatically retrieve player IP and profile information.

Remarks

The **peerAlwaysGetPlayerInfo** function may be used to tell the sdk the automatically retrieve player IP and profile information when joining a room.

Section Reference: [Gamespy Peer SDK](#)

peerAreAllReady

Used to check if all players in the staging room are ready.

```
PEERBool peerAreAllReady(  
    PEER peer );
```

Routine	Required Header	Distribution
peerAreAllReady	<peer.h>	SDKZIP

Return Value

Returns PEERTrue if all players are ready. Otherwise returns PEERFalse.

Parameters

peer

[in] Initialized peer object.

Remarks

The **peerAreAllReady** function may be used to determine if all players are ready. This is generally used before `peerStartGame`.

Section Reference: [Gamespy Peer SDK](#)

peerAuthenticateCDKey

Allows pre-chat cd key authentication via the chat server.

```
void peerAuthenticateCDKey(  
    PEER peer,  
    const gsi_char * cdkey,  
    peerAuthenticateCDKeyCallback callback,  
    void * param,  
    PEERBool blocking );
```

Routine	Required Header	Distribution
peerAuthenticateCDKey	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object.

cdkey

[in] The cdkey to validate. Presumably a valid cdkey for the set game title.

callback

[in] Callback function will be called when the operation has completed.

param

[in] Pointer to user data. This is optional and will be passed unmodified to the callback function.

blocking

[in] When set to PEERTrue this function will not return until the operation has completed.

Remarks

The **peerAuthenticateCDKey** function may be used to authenticate a user's cdkey before they enter the chat room. This should not be a substitute for a cdkey during gameplay. Arcade does not support this call, so users in Arcade will be able to enter chat without this validation. This method is most useful for developers who opt-out of the Arcade compatibility requirements or have a separate chat area for in-game clients.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
peerAuthenticateCDKey	peerAuthenticateCDKeyA	peerAuthenticateCDKeyW

peerAuthenticateCDKeyW and **peerAuthenticateCDKeyA** are UNICODE and ANSI mapped versions of **peerAuthenticateCDKey**. The arguments of **peerAuthenticateCDKeyA** are ANSI strings; those of **peerAuthenticateCDKeyW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

peerChangeNick

Change the chat nickname associated with the local client. This does not affect the account name.

```
void peerChangeNick(  
    PEER peer,  
    const gsi_char * newNick,  
    peerChangeNickCallback callback,  
    void * param,  
    PEERBool blocking );
```

Routine	Required Header	Distribution
peerChangeNick	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object.

newNick

[in] The nickname to assign to the local user.

callback

[in] Callback function will be called when the operation completes.

param

[in] Pointer to user data. This is optional and will be passed unmodified to the callback function.

blocking

[in] When set to PEERTrue this function will not return until the operation has completed.

Remarks

The **peerChangeNick** function may be used to change a user's nickname as it appears in chat. This has no affect on GameSpy profile names such as those used for presence detection and buddy lists. Only one instance of a nickname may be in use at a time. The attempt may fail if the nick is invalid or in use. This will fail if Peer is not connected.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
<code>peerChangeNick</code>	<code>peerChangeNickA</code>	<code>peerChangeNickW</code>

peerChangeNickW and **peerChangeNickA** are UNICODE and ANSI mapped versions of **peerChangeNick**. The arguments of **peerChangeNickA** are ANSI strings; those of **peerChangeNickW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

peerClearTitle

Resets the peer SDK. peerSetTitle must be called before new operations are made.

```
void peerClearTitle(  
    PEER peer );
```

Routine	Required Header	Distribution
peerClearTitle	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object.

Remarks

The **peerClearTitle** function resets the peer sdk to initialized state. Until a title is set again with `peerSetTitle`, the Peer SDK will be unable to join rooms, list games, etc.

Section Reference: [Gamespy Peer SDK](#)

peerConnect

Connect to the chat server.

```
void peerConnect(  
    PEER peer,  
    const gsi_char * nick,  
    int profileID,  
    peerNickErrorCallback nickErrorCallback,  
    peerConnectCallback connectCallback,  
    void * param,  
    PEERBool blocking );
```

Routine	Required Header	Distribution
peerConnect	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object.

nick

[in] Chat nickname.

profileID

[in] Profile ID of the local client, or 0.

nickErrorCallback

[in] Callback function is called if the nickname is invalid or is already being used.

connectCallback

[in] Callback function is called when the connect operation is completed.

param

[in] Pointer to user data. This is optional and will be passed unmodified to the callback function.

blocking

[in] When set to PEERTrue this function will not return until the operation has completed.

Remarks

The **peerConnect** function opens a connection to the chat server. The connection can be ended at any time by calling `peerDisconnect()` (`peerShutdown()` will also close the connection). If the connection gets disconnected for any other reason (such as an intermediate router going down), the `peerDisconnectedCallback()` callback will be called. This function will fail if the Peer object is already connected.

Once connected to the backend chat server, Peer is fully enabled, and the user can join rooms, create rooms, list games, message other players, etc. Typically at this point the user would be joined up to the game's title room with `peerJoinTitleRoom()`.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
peerConnect	peerConnectA	peerConnectW

peerConnectW and **peerConnectA** are UNICODE and ANSI mapped versions of **peerConnect**. The arguments of **peerConnectA** are ANSI strings; those of **peerConnectW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

peerConnectLogin

Connects a peer object to the backend chat server and then logs in using an account from the GameSpy Presence system. A title must be set with `peerSetTitle()` before this function is called.

```
void peerConnectLogin(  
    PEER peer,  
    int namespaceID,  
    const gsi_char * email,  
    const gsi_char * profilenick,  
    const gsi_char * uniquenick,  
    const gsi_char * password,  
    peerNickErrorCallback nickErrorCallback,  
    peerConnectCallback connectCallback,  
    void * param,  
    PEERBool blocking );
```

Routine	Required Header	Distribution
peerConnectLogin	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object.

namespaceID

[in] Namespace identifier, assigned by GameSpy.

email

[in] Email address of the profile to login with.

profilenick

[in] Nickname of the profile to login with.

uniquenick

[in] Registered uniquenick of the profile to login with.

password

[in] Password of the profile to login with.

nickErrorCallback

[in] Callback function to be called if a nickname error occurs.

connectCallback

[in] Callback function to be called when the operation completes

param

[in] Pointer to user data. This is optional and will be passed unmodified to the callback function.

blocking

[in] When set to PEERTrue this function will not return until the operation has completed.

Remarks

The **peerConnectLogin** function attempts to connect a peer object to the backend chat server and then login using an account from the GameSpy Presence system.. The connection can be ended at any time by called `peerDisconnect()` (`peerShutdown()` will also close the connection). If the connection gets disconnected for any other reason (such as an intermediate router going down), the `peerDisconnectedCallback()` callback will be called. This function will fail if the Peer object is already connected.

Once connected to the backend chat server, Peer is fully enabled, and the user can join rooms, create rooms, list games, message other players, etc. Typically at this point the user would be joined up to the game's title room with `peerJoinTitleRoom()`.

There are two ways of specifying the account information. One way is to specify a uniquenick and password combination. In this case, email and profilenick should be NULL, and namespaceID should be greater than 0. The other way is to specify an email, profilenick, and password. In this case, uniquenick should be NULL, and namespaceID should be 0 for no namespace or 1 for the default GameSpy namespace (used by GameSpy Arcade). If you are using a custom namespace, specify its namespace ID. You can contact GameSpy Developer Support for further help in using a custom namespace.

The uniquenick and password combination should only be used in custom namespaces where uniquenicks do not expire. This is because if a uniquenick expires, then another user that registers that uniquenick, the original user will no longer be able to login with only his (old) uniquenick and password.

Section Reference: [Gamespy Peer SDK](#)

peerConnectPreAuth

Connects a peer object to the backend chat server and then logs in using authentication information from a partner authentication system.

```
void peerConnectPreAuth(  
    PEER peer,  
    const gsi_char * authtoken,  
    const gsi_char * partnerchallenge,  
    peerNickErrorCallback nickErrorCallback,  
    peerConnectCallback connectCallback,  
    void * param,  
    PEERBool blocking );
```

Routine	Required Header	Distribution
peerConnectPreAuth	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object.

authtoken

[in] Authtoken for this login.

partnerchallenge

[in] Partner challenge for this login.

nickErrorCallback

[in] Callback function to be called if a nick error occurs.

connectCallback

[in] Callback function to be called when the operation completes.

param

[in] Pointer to user data. This is optional and will be passed unmodified to the callback function.

blocking

[in] When set to PEERTrue this function will not return until the operation has completed.

Remarks

The **peerConnectPreAuth** attempts to connect a peer object to the backend chat server and then login using an account from a partner authentication system. The connection can be ended at any time by called `peerDisconnect()` (`peerShutdown()` will also close the connection). If the connection gets disconnected for any other reason (such as an intermediate router going down), the `peerDisconnectedCallback()` callback will be called. This function will fail if the Peer object is already connected.

Once connected to the backend chat server, Peer is fully enabled, and the user can join rooms, create rooms, list games, message other players, etc. Typically at this point the user would be joined up to the game's title room with `peerJoinTitleRoom()`. A title must be set with `peerSetTitle()` before this function is called.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
peerConnectPreAuth	peerConnectPreAuthA	peerConnectPreAuthW

peerConnectPreAuthW and **peerConnectPreAuthA** are UNICODE and ANSI mapped versions of **peerConnectPreAuth**. The arguments of **peerConnectPreAuthA** are ANSI strings; those of **peerConnectPreAuthW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

See Also: [peerConnect](#), [peerConnectLogin](#)

peerCreateStagingRoom

Creates a new staging room with the local player as the host.

```
void peerCreateStagingRoom(  
    PEER peer,  
    const gsi_char * name,  
    int maxPlayers,  
    const gsi_char password[PEER_PASSWORD_LEN],  
    peerJoinRoomCallback callback,  
    void * param,  
    PEERBool blocking );
```

Routine	Required Header	Distribution
peerCreateStagingRoom	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object.

name

[in] Staging room name.

maxPlayers

[in] Maximum number of players allowed in the room.

password

[in] Optional room password.

callback

[in] Callback function to be called when the operation completes.

param

[in] Pointer to user data. This is optional and will be passed unmodified to the callback function.

blocking

[in] When set to PEERTrue, this call will not return until the operation has completed.

Remarks

The **peerCreateStagingRoom** function creates a new staging room with the local client as the host. Staging room names are not unique and multiple staging rooms may have the same name. If the password parameter is not NULL or "", this will create a passworded room. The same case-sensitive password needs to be passed into `peerJoinStagingRoom[ByIP]()` for other player's to join the room. Spaces in passwords are not allowed. Any password with spaces should be stripped of those spaces before calling this function, or a warning to the user will suffice. No more than `maxPlayers` players will be allowed in the room, unless `maxPlayers` is set to 0, in which case no limit is set and the `maxplayers` key is not reported. If the user is in a group room when this function is called, then the room will be reported as being a part of that group (even if the local user then leaves and joins another group). If successful, the `peerQR*Callback()` callbacks will start getting called. These are used to provide information about the room to other players. For more information on what to report in the callbacks, see their descriptions. This function is only valid if a title is set. If the user is already in a staging room, this function will fail.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defi
peerCreateStagingRoom	peerCreateStagingRoomA	peerCreateStagingR

peerCreateStagingRoomW and **peerCreateStagingRoomA** are UNICODE and ANSI mapped versions of **peerCreateStagingRoom**. The arguments of **peerCreateStagingRoomA** are ANSI strings; those of **peerCreateStagingRoomW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

See Also: [peerJoinStagingRoom](#)

peerCreateStagingRoomWithSocket

Creates a new staging room with the local player as the host.

```
void peerCreateStagingRoomWithSocket(  
    PEER peer,  
    const gsi_char * name,  
    int maxPlayers,  
    const gsi_char password[PEER_PASSWORD_LEN],  
    SOCKET socket,  
    unsigned short port,  
    peerJoinRoomCallback callback,  
    void * param,  
    PEERBool blocking );
```

Routine	Required Header	Distribution
peerCreateStagingRoomWithSocket	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object.

name

[in] The name of the staging room.

maxPlayers

[in] Optional max number of players allowed in the room. May be 0.

password

[in] Optional room password.

socket

[in] The socket that is being shared. Parameter used when calling `peerCreateStagingRoomWithSocket`.

port

[in] The local port the socket is bound to. Parameter used when calling `peerCreateStagingRoomWithSocket`.

callback

[in] Callback function called when the operation completes.

param

[in] Pointer to user data. This is optional and will be passed unmodified to the callback function.

blocking

[in] When set to `PEERTrue`, this call will not return until the operation has completed.

Remarks

The **peerCreateStagingRoomWithSocket** creates a new staging room, with the local player hosting. If the password parameter is not NULL or "", this will create a passworded room. The same case-sensitive password needs to be passed into `peerJoinStagingRoom[ByIP]()` for other player's to join the room. Spaces in passwords are not allowed. Any password with spaces should be stripped of those spaces before calling this function, or a warning to the user will suffice. No more than `maxPlayers` players will be allowed in the room, unless `maxPlayers` is set to 0, in which case no limit is set and the `maxplayers` key is not reported. If the user is in a group room when this function is called, then the room will be reported as being a part of that group (even if the local user then leaves and joins another group).

If successful, the `peerQR*Callback()` callbacks will start getting called. These are used to provide information about the room to other players. For more information on what to report in the callbacks, see their descriptions.

If **peerCreateStagingRoomWithSocket** is used, the socket provided must be an already created UDP socket. It will be used for sending out query replies, and any queries the application reads off of the socket must be passed to Peer using `peerParseQuery()`. This can be useful when running a game host behind a NAT/firewall/proxy--for a full explanation of how this helps, see the "NAT and Firewall Support" appendix in the "GameSpy Query and Reporting 2 SDK" documentation. This function is only valid if a title is set. If the user is already in a staging room, this function will fail.

Section Reference: [Gamespy Peer SDK](#)

peerDisconnect

Disconnect from the chat server. `peerShutdown` must still be called.

```
void peerDisconnect(  
    PEER peer );
```

Routine	Required Header	Distribution
peerDisconnect	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object.

Remarks

The **peerDisconnect** function disconnects the local client from the chat server. `peerShutdown` must still be called to free internal sdk memory.

Section Reference: [Gamespy Peer SDK](#)

peerEnumPlayers

Enumerates through the players in a room.

```
void peerEnumPlayers(  
    PEER peer,  
    RoomType roomType,  
    peerEnumPlayersCallback callback,  
    void * param );
```

Routine	Required Header	Distribution
peerEnumPlayers	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object.

roomType

[in] Can be either TitleRoom, GroupRoom or StagingRoom.

callback

[in] Callback function to be called when the operation completes.

param

[in] Pointer to user data. This is optional and will be passed unmodified to the callback function.

Remarks

The **peerEnumPlayers** function may be used to iterate through the list of players in the specified room type. The callback will be called for each player in the room, and then once again when done with "index" set to -1 and "nick" set to NULL. The enumeration is done using a local list of players, and the callbacks will be called from within the function call. This only works if there is a title set, and the user is in the room.

Section Reference: [Gamespy Peer SDK](#)

peerFixNick

Repairs an illegal chat nickname. Removes illegal characters from a nickname as well as invalid character combinations.

```
void peerFixNick(  
    gsi_char * newNick,  
    const gsi_char * oldNick );
```

Routine	Required Header	Distribution
peerFixNick	<peer.h>	SDKZIP

Parameters

newNick

[out] Corrected nickname. May be the same as oldNick if no issues are detected.

oldNick

[in] The nickname to be corrected or verified.

Remarks

The **peerFixNick** function replaces illegal characters in the nickname with the underscore ("_") character. This function will also replace leading numbers and illegal whitespace combinations. Because of the possibility of an underscore being added to the beginning, newNick should be able to hold at least one character more than oldNick.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
peerFixNick	peerFixNickA	peerFixNickW

peerFixNickW and **peerFixNickA** are UNICODE and ANSI mapped versions of **peerFixNick**. The arguments of **peerFixNickA** are ANSI strings; those of **peerFixNickW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

peerGetAutoMatchStatus

Used when automatching to retrieve the current status.

```
PEERAutoMatchStatus peerGetAutoMatchStatus(  
    PEER peer );
```

Routine	Required Header	Distribution
peerGetAutoMatchStatus	<peer.h>	SDKZIP

Return Value

The current auto match status. See remarks.

Parameters

peer

[in] Initialized peer object.

Remarks

The **peerGetAutoMatchStatus** function returns the current automatch status. The return value is one of the PEERAutoMatchStatus enumerated types.

Section Reference: [Gamespy Peer SDK](#)

peerGetChat

Returns the chat sdk object.

CHAT peerGetChat(
PEER peer);

Routine	Required Header	Distribution
peerGetChat	<peer.h>	SDKZIP

Return Value

The chat sdk object being used by the peer sdk.

Parameters

peer

[in] Initialized peer object.

Remarks

The **peerGetChat** function returns the chat sdk object being used by the peer sdk. The peer sdk wraps the chat sdk. An application can use this function to get a reference to the chat object, which allows it to directly access some of the Chat SDK's lower level functionality. For example, this could be used to join a separate chat channel, outside of the scope of Peer. This function will fail if Peer is not yet connected to the chat server. The chat object will become invalid as soon as the peer object is disconnected.

Section Reference: [Gamespy Peer SDK](#)

peerGetGlobalWatchKey

Returns the cached value of a players watch key.

```
const gsi_char * peerGetGlobalWatchKey(  
    PEER peer,  
    const gsi_char * nick,  
    const gsi_char * key );
```

Routine	Required Header	Distribution
peerGetGlobalWatchKey	<peer.h>	SDKZIP

Return Value

The watch key's value, or NULL if the watch key is unknown. (Empty string "" is a legal value).

Parameters

peer

[in] Initialized peer object.

nick

[in] The nickname of the player.

key

[in] The name of the key.

Remarks

The **peerGetGlobalWatchKey** function may be used to retrieve the cached value of a global watch key. If the key value is not known or has not been received this function will return NULL. Please note that an empty string "" is valid key value. If the key is just empty (or was never set), an empty string will be returned.

The key being requested must have previously been set as a global watch key, with **peerSetGlobalWatchKey()**, for a room that the player and the local player have in common. This will fail if no title is set.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defi
peerGetGlobalWatchKey	peerGetGlobalWatchKeyA	peerGetGlobalWatch

peerGetGlobalWatchKeyW and **peerGetGlobalWatchKeyA** are UNICODE and ANSI mapped versions of **peerGetGlobalWatchKey**. The arguments of **peerGetGlobalWatchKeyA** are ANSI strings; those of **peerGetGlobalWatchKeyW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

peerGetGroupID

Returns the current group ID set from peerJoinGroupRoom or peerSetGroupID.

```
int peerGetGroupID(  
    PEER peer );
```

Routine	Required Header	Distribution
peerGetGroupID	<peer.h>	SDKZIP

Return Value

The current group ID, otherwise a zero is returned.

Parameters

peer

[in] Initialized peer object.

Remarks

The **peerGetGroupID** function returns the current group id. This id may be set automatically when joining a room with `peerJoinGroupRoom` or manually using `peerSetGroupID`. The peer object in this function needs to be initialized and connected otherwise a zero is returned.

Section Reference: [Gamespy Peer SDK](#)

peerGetHostServer

Returns the SBServer object associated with the staging room host.

```
SBServer peerGetHostServer(  
    PEER peer );
```

Routine	Required Header	Distribution
peerGetHostServer	<peer.h>	SDKZIP

Return Value

Returns the SBServer object associated with the staging room host.
NULL if the local client is not a member of a staging room.

Parameters

peer

[in] Initialized peer object.

Remarks

The **peerGetHostServer** function returns the SBServer object associated with the local host. Information on the host may be retrieved using the SBServer data accessors.

Section Reference: [Gamespy Peer SDK](#)

peerGetNick

Returns the chat nickname of the local client.

```
const gsi_char * peerGetNick(  
    PEER peer );
```

Routine	Required Header	Distribution
peerGetNick	<peer.h>	SDKZIP

Return Value

The chat nickname of the local user. NULL if not connected.

Parameters

peer

[in] Initialized peer object.

Remarks

The **peerGetNick** function returns the chat nickname of the local client. The peer object must be initialized and connected to a chat server. Otherwise a NULL is returned if not connected.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
peerGetNick	peerGetNickA	peerGetNickW

peerGetNickW and **peerGetNickA** are UNICODE and ANSI mapped versions of **peerGetNick**. The arguments of **peerGetNickA** are ANSI strings; those of **peerGetNickW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

peerGetPlayerFlags

Returns the cached flag values for the current player. This is from the key "b_flags".

```
PEERBool peerGetPlayerFlags(  
    PEER peer,  
    const gsi_char * nick,  
    RoomType roomType,  
    int * flags );
```

Routine	Required Header	Distribution
peerGetPlayerFlags	<peer.h>	SDKZIP

Return Value

This function returns PEERTrue for success. PEERFalse otherwise.

Parameters

peer

[in] Initialized peer object.

nick

[in] The players chat nickname.

roomType

[in] Can be either TitleRoom, GroupRoom or StagingRoom.

flags

[out] The player flags.

Remarks

The **peerGetPlayerFlags** function will return the cached flag values for the specified player. A server query is not sent at this time. This function will return `PEERFalse` if the player is not in the specified room. The flags are a per-room setting. That is, if the local player is in multiple rooms with another player, that other player may have different flags in each of the rooms.

Flags might not be available for a player that just joined, or if the local player just joined the room. Also, flags might not be available for players that aren't using the Peer SDK. However, this function will not return false in that case. Instead, it will just set the flags to empty.

The flags each represent one bit in the "flags" integer. The flags are:

`PEER_FLAG_STAGING`: the player is in a staging room.

`PEER_FLAG_READY`: the player is readied up for a game.

`PEER_FLAG_PLAYING`: the player is playing a game.

`PEER_FLAG_AWAY`: the player is away.

`PEER_FLAG_HOST`: the player is the host of the room.

`PEER_FLAG_OP`: the player is an op (+o) in this room.

`PEER_FLAG_VOICE`: the player has voice (+v) in this room.

This function will fail if no title is set.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
peerGetPlayerFlags	peerGetPlayerFlagsA	peerGetPlayerFlagsW

peerGetPlayerFlagsW and **peerGetPlayerFlagsA** are UNICODE and ANSI mapped versions of **peerGetPlayerFlags**. The arguments of **peerGetPlayerFlagsA** are ANSI strings; those of **peerGetPlayerFlagsW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

peerGetPlayerGlobalKeys

Query the server for a player's global key values.

```
void peerGetPlayerGlobalKeys(  
    PEER peer,  
    const gsi_char * nick,  
    int num,  
    const gsi_char ** keys,  
    peerGetGlobalKeysCallback callback,  
    void * param,  
    PEERBool blocking );
```

Routine	Required Header	Distribution
peerGetPlayerGlobalKeys	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object.

nick

[in] The chat nickname of the target player.

num

[in] The number of keys in the array.

keys

[in] Array of key names to request values for.

callback

[in] Callback function to be called when the operation completes.

param

[in] Pointer to user data. This is optional and will be passed unmodified to the callback function.

blocking

[in] When set to PEERTrue this function will not return until the operation has completed.

Remarks

The **peerGetPlayerGlobalKeys** function may be used to retrieve global key values for the specified player. The callback will have these keys available if the function is successful. The key list will be the array of strings that are used to obtain the global key values.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE D
peerGetPlayerGlobalKeys	peerGetPlayerGlobalKeysA	peerGetPlayerGlo

peerGetPlayerGlobalKeysW and **peerGetPlayerGlobalKeysA** are UNICODE and ANSI mapped versions of **peerGetPlayerGlobalKeys**. The arguments of **peerGetPlayerGlobalKeysA** are ANSI strings; those of **peerGetPlayerGlobalKeysW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

peerGetPlayerInfo

Retrieve a local player's IP and profile ID.

```
void peerGetPlayerInfo(  
    PEER peer,  
    const gsi_char * nick,  
    peerGetPlayerInfoCallback callback,  
    void * param,  
    PEERBool blocking );
```

Routine	Required Header	Distribution
peerGetPlayerInfo	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object.

nick

[in] Chat nickname of the target player.

callback

[in] Callback function to be called when the operation completes.

param

[in] Pointer to user data. This is optional and will be passed unmodified to the callback function.

blocking

[in] When set to PEERTrue this function will not return until the operation has completed.

Remarks

The **peerGetPlayerInfo** function queries the chat server for the local player's IP and profile ID. The information will be available once the function is successful and the callback gets called. The callback will have both the profileID and IP address of the local player.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
peerGetPlayerInfo	peerGetPlayerInfoA	peerGetPlayerInfoW

peerGetPlayerInfoW and **peerGetPlayerInfoA** are UNICODE and ANSI mapped versions of **peerGetPlayerInfo**. The arguments of **peerGetPlayerInfoA** are ANSI strings; those of **peerGetPlayerInfoW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

See Also: [peerGetPlayerInfoCallback](#)

peerGetPlayerInfoNoWait

Retrieve a players IP and profile ID. Uses a cached copy.

```
PEERBool peerGetPlayerInfoNoWait(  
    PEER peer,  
    const gsi_char * nick,  
    unsigned int * IP,  
    int * profileID );
```

Routine	Required Header	Distribution
peerGetPlayerInfoNoWait	<peer.h>	SDKZIP

Return Value

Returns PEERTrue if a cached copy was available. PEERFalse otherwise.

Parameters

peer

[in] Initialized peer object.

nick

[in] Chat nickname of the target player.

IP

[out] IP address of the target player.

profileID

[out] Profile ID of the target player.

Remarks

The **peerGetPlayerInfoNoWait** function returns the cached copy of the player's IP and profile ID. Use in conjunction with **peerAlwaysGetPlayerInfo**. Returns **PEERFalse** if the info is not available. Reasons why the info would not be immediately available include the local player not being in any common room with the player, the player is not using the Peer SDK, or we have just joined a room and we don't yet have everyone's info.

Even if this succeeds, the profile ID can be 0 if it's not available. This function will fail if not connected.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE De
peerGetPlayerInfoNoWait	peerGetPlayerInfoNoWaitA	peerGetPlayerInfoN

peerGetPlayerInfoNoWaitW and **peerGetPlayerInfoNoWaitA** are UNICODE and ANSI mapped versions of **peerGetPlayerInfoNoWait**. The arguments of **peerGetPlayerInfoNoWaitA** are ANSI strings; those of **peerGetPlayerInfoNoWaitW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

peerGetPlayerPing

Returns the cached ping between the local player and the specified remote player.

```
PEERBool peerGetPlayerPing(  
    PEER peer,  
    const gsi_char * nick,  
    int * ping );
```

Routine	Required Header	Distribution
peerGetPlayerPing	<peer.h>	SDKZIP

Return Value

This function return PEERTrue if a cached ping time was available.
PEERFalse otherwise.

Parameters

peer

[in] Initialized peer object.

nick

[in] Nick of the target player.

ping

[out] This will be set to the cached ping time.

Remarks

The **peerGetPlayerPing** function is used to retrieve the cached ping value for the specified remote player. Returns PEERFalse if we don't have or can't get a ping for this player. This function only works if a title is set.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
peerGetPlayerPing	peerGetPlayerPingA	peerGetPlayerPingW

peerGetPlayerPingW and **peerGetPlayerPingA** are UNICODE and ANSI mapped versions of **peerGetPlayerPing**. The arguments of **peerGetPlayerPingA** are ANSI strings; those of **peerGetPlayerPingW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

peerGetPlayersCrossPing

Calculates the cross-ping between 2 players.

```
PEERBool peerGetPlayersCrossPing(  
    PEER peer,  
    const gsi_char * nick1,  
    const gsi_char * nick2,  
    int * crossPing );
```

Routine	Required Header	Distribution
peerGetPlayersCrossPing	<peer.h>	SDKZIP

Return Value

This function returns PEERTrue if a cached cross ping is available.
PEERFalse otherwise.

Parameters

peer

[in] Initialized peer object.

nick1

[in] Chat nickname of player 1.

nick2

[in] Chat nickname of player 2.

crossPing

[out] This is set to the cross-ping, if available.

Remarks

The **peerGetPlayersCrossPing** function is used to calculate the cross ping between two players. Returns PEERFalse if we don't have or can't get the player's cross-ping. The ordering of the nicks does not matter (i.e., peer stores the pings between sets of players, not each player's ping to each other player). This function only works if a title is set, and the peer object is connected to the chat server.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE D
peerGetPlayersCrossPing	peerGetPlayersCrossPingA	peerGetPlayersCr

peerGetPlayersCrossPingW and **peerGetPlayersCrossPingA** are UNICODE and ANSI mapped versions of **peerGetPlayersCrossPing**. The arguments of **peerGetPlayersCrossPingA** are ANSI strings; those of **peerGetPlayersCrossPingW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

peerGetPrivateIP

Returns the local private IP address.

```
unsigned int peerGetPrivateIP(  
    PEER peer );
```

Routine	Required Header	Distribution
peerGetPrivateIP	<peer.h>	SDKZIP

Return Value

Returns the local private IP address.

Parameters

peer

[in] Initialized peer object.

Remarks

The **peerGetPrivateIP** function returns the local private IP address. If called while not connected, or if there is no private address, will return 0. A private address is any local IP in a private IP range. The IP masks for these ranges (as specified in RFC 1918) are 10.*, 172.16-31.*, and 192.168.*.

Section Reference: [Gamespy Peer SDK](#)

peerGetProfileID

Returns the profile ID of the local client. Only valid with peerConnectLogin or peerConnectPreAuth.

```
int peerGetProfileID(  
    PEER peer );
```

Routine	Required Header	Distribution
peerGetProfileID	<peer.h>	SDKZIP

Return Value

The profile ID of the local client.

Parameters

peer

[in] Initialized peer object.

Remarks

The **peerGetProfileID** function returns the profile ID of the local client. This uniquely identifies a profile (nick/email/password or uniquenick/password). See the Presence SDK documentation for more details.

Section Reference: [Gamespy Peer SDK](#)

peerGetPublicIP

Returns the local public IP address.

```
unsigned int peerGetPublicIP(  
    PEER peer );
```

Routine	Required Header	Distribution
peerGetPublicIP	<peer.h>	SDKZIP

Return Value

Returns the local public IP address.

Parameters

peer

[in] Initialized peer object.

Remarks

The **peerGetPublicIP** function returns the local public IP address. If called while not connected, will return 0. The IP this returns is the externally visible IP (e.g. for NATs).

Section Reference: [Gamespy Peer SDK](#)

See Also: [peerGetPrivateIP](#)

peerGetReady

Get the ready state of the specified player.

```
PEERBool peerGetReady(  
    PEER peer,  
    const gsi_char * nick,  
    PEERBool * ready );
```

Routine	Required Header	Distribution
peerGetReady	<peer.h>	SDKZIP

Return Value

This function returns PEERTrue if the ready state is available.
PEERFalse otherwise.

Parameters

peer

[in] Initialized peer object.

nick

[in] Nickname of the target player.

ready

[out] Set to PEERTrue if the player is ready, PEERFalse if the player is not.

Remarks

The **peerGetReady** function may be used to determine the ready status of each player in a staging room. This is often useful for display an icon or informational message to the host. This is only valid when in a staging room.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
peerGetReady	peerGetReadyA	peerGetReadyW

peerGetReadyW and **peerGetReadyA** are UNICODE and ANSI mapped versions of **peerGetReady**. The arguments of **peerGetReadyA** are ANSI strings; those of **peerGetReadyW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

peerGetRoomChannel

Returns the chat channel associated with the room type.

```
const gsi_char * peerGetRoomChannel(  
    PEER peer,  
    RoomType roomType );
```

Routine	Required Header	Distribution
peerGetRoomChannel	<peer.h>	SDKZIP

Return Value

This function returns the chat channel name for the specified room type.
NULL if the local client is not a member of the specified room type.

Parameters

peer

[in] Initialized peer object.

roomType

[in] Can be either TitleRoom, GroupRoom or StagingRoom.

Remarks

The **peerGetRoomChannel** gets the chat channel associated with the room. It returns NULL if not in the room.

Section Reference: [Gamespy Peer SDK](#)

peerGetRoomGlobalKeys

Retrieves global keys for all players in the specified room. (Local client must be a room member.).

```
void peerGetRoomGlobalKeys(  
    PEER peer,  
    RoomType roomType,  
    int num,  
    const gsi_char ** keys,  
    peerGetGlobalKeysCallback callback,  
    void * param,  
    PEERBool blocking );
```

Routine	Required Header	Distribution
peerGetRoomGlobalKeys	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object.

roomType

[in] Can be either TitleRoom, GroupRoom or StagingRoom.

num

[in] Number of keys in the array parameter - keys.

keys

[in] Array of key names to retrieve values for.

callback

[in] Callback function to be called when the operation completes.

param

[in] Pointer to user data. This is optional and will be passed unmodified to the callback function.

blocking

[in] When set to PEERTrue this function will not return until the operation has completed.

Remarks

The **peerGetRoomGlobalKeys** function retrieves the global keys for all players in a room we're in. The callback will be called once for each player in the room, then once more with "nick" set to NULL. This will fail if no title is set, and the peer object is not connected.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE De
peerGetRoomGlobalKeys	peerGetRoomGlobalKeysA	peerGetRoomGlob

peerGetRoomGlobalKeysW and **peerGetRoomGlobalKeysA** are UNICODE and ANSI mapped versions of **peerGetRoomGlobalKeys**. The arguments of **peerGetRoomGlobalKeysA** are ANSI strings; those of **peerGetRoomGlobalKeysW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

See Also: [peerGetPlayerGlobalKeys](#)

peerGetRoomKeys

Retrieves room key values for the room or a single player.

```
void peerGetRoomKeys(  
    PEER peer,  
    RoomType roomType,  
    const gsi_char * nick,  
    int num,  
    const gsi_char ** keys,  
    peerGetRoomKeysCallback callback,  
    void * param,  
    PEERBool blocking );
```

Routine	Required Header	Distribution
peerGetRoomKeys	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object.

roomType

[in] Can be either TitleRoom, GroupRoom or StagingRoom.

nick

[in] Nickname of the player to retrieve values on. "*" to retrieve values for the entire room.

num

[in] Number of valid key names in the array parameter - keys.

keys

[in] Array of key names to retrieve values for.

callback

[in] Callback function to be called when the operation completes.

param

[in] Pointer to user data. This is optional and will be passed unmodified to the callback function.

blocking

[in] When set to PEERTrue, this function will not return until the operation has completed.

Remarks

The **peerGetRoomKeys** function retrieves the keys for either a room, a player in a room, or all the players in a room. If getting keys for a room, or for a single player, the callback will be called once. If getting keys for every player in a room, then it will be called once for each player, then one more time with nick set to NULL. This will fail if no title is set, and the peer object is not connected.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
peerGetRoomKeys	peerGetRoomKeysA	peerGetRoomKeysW

peerGetRoomKeysW and **peerGetRoomKeysA** are UNICODE and ANSI mapped versions of **peerGetRoomKeys**. The arguments of **peerGetRoomKeysA** are ANSI strings; those of **peerGetRoomKeysW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

peerGetRoomName

Returns the channel's title. The local client must be a member of the room.

```
const gsi_char * peerGetRoomName(  
    PEER peer,  
    RoomType roomType );
```

Routine	Required Header	Distribution
peerGetRoomName	<peer.h>	SDKZIP

Return Value

Returns the channel's title. NULL if the local client is not a member of the specified room type.

Parameters

peer

[in] Initialized peer object.

roomType

[in] Can be either TitleRoom, GroupRoom or StagingRoom.

Remarks

The **peerGetRoomName** function retrieves the name of the room the local player is in. It return NULL if not in the room.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
<code>peerGetRoomName</code>	<code>peerGetRoomNameA</code>	<code>peerGetRoomNameW</code>

peerGetRoomNameW and **peerGetRoomNameA** are UNICODE and ANSI mapped versions of **peerGetRoomName**. The arguments of **peerGetRoomNameA** are ANSI strings; those of **peerGetRoomNameW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

peerGetTitle

Gets the currently set title.

```
const gsi_char * peerGetTitle(  
    PEER peer );
```

Routine	Required Header	Distribution
peerGetTitle	<peer.h>	SDKZIP

Return Value

Parameters

peer

[in] Initialized peer object.

Remarks

The **peerGetTitle** function retrieves the currently set title. It returns NULL if there is no title set.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
peerGetTitle	peerGetTitleA	peerGetTitleW

peerGetTitleW and **peerGetTitleA** are UNICODE and ANSI mapped versions of **peerGetTitle**. The arguments of **peerGetTitleA** are ANSI strings; those of **peerGetTitleW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

peerGetUserID

Returns the local userID. Only valid with peerConnectLogin or peerConnectPreAuth.

```
int peerGetUserID(  
    PEER peer );
```

Routine	Required Header	Distribution
peerGetUserID	<peer.h>	SDKZIP

Return Value

Returns the local userID.

Parameters

peer

[in] Initialized peer object.

Remarks

The **peerGetUserID** function retrieves the local user's userID. This uniquely identifies a user account (email and password combination). See the Presence SDK documentation for more details.

Section Reference: [Gamespy Peer SDK](#)

peerInitialize

Initialize the peer SDK.

```
PEER peerInitialize(  
    PEERCallbacks * callbacks );
```

Routine	Required Header	Distribution
peerInitialize	<peer.h>	SDKZIP

Return Value

Initialized peer sdk.

Parameters

callbacks

[in] PEERCallbacks structure filled with appropriate callbacks.

Remarks

The **peerInitialize** function creates a peer object. This object is valid until it is passed to `peerShutdown()`. After initialization, the next two steps will usually be to set the title with `peerSetTitle()` then connect with `peerConnect()`. It is valid to have multiple Peer objects allocated at any given time, however this is usually not needed.

Section Reference: [Gamespy Peer SDK](#)

peerInRoom

Determines if the local client is in a room of the specified type.

```
PEERBool peerInRoom(  
    PEER peer,  
    RoomType roomType );
```

Routine	Required Header	Distribution
peerInRoom	<peer.h>	SDKZIP

Return Value

This function returns PEERTrue if the local client is in the room, PEERFalse otherwise.

Parameters

peer

[in] Initialized peer object.

roomType

[in] Can be either TitleRoom, GroupRoom or StagingRoom.

Remarks

The **peerInRoom** function checks whether or not the local player is in the specified room type. It returns PEERTrue if the localy player is in the room specified.

Section Reference: [Gamespy Peer SDK](#)

peerIsAutoMatching

Returns PEERTrue if an AutoMatch is in progress.

```
PEERBool peerIsAutoMatching(  
    PEER peer );
```

Routine	Required Header	Distribution
peerIsAutoMatching	<peer.h>	SDKZIP

Return Value

Returns PEERTrue if an AutoMatch is in progress, PEERFalse otherwise.

Parameters

peer

[in] Initialized peer object.

Remarks

The **peerIsAutoMatching** function is used to determine if an AutoMatch is in progress. Returns PEERTrue if an AutoMatch is in progress, PEERFalse if otherwise.

Section Reference: [Gamespy Peer SDK](#)

peerIsConnected

Returns PEERTrue if connected to the chat server.

```
PEERBool peerIsConnected(  
    PEER peer );
```

Routine	Required Header	Distribution
peerIsConnected	<peer.h>	SDKZIP

Return Value

Returns PEERTrue if connected to the chat server, PEERFalse otherwise.

Parameters

peer

[in] Initialized peer object.

Remarks

The **peerIsConnected** function tests to check whether the peer object has connected to the chat server. It returns PEERFalse if the peer object has not connected.

Section Reference: [Gamespy Peer SDK](#)

peerIsPlayerHost

Returns PEERTrue if specified player is a room host or operator.

```
PEERBool peerIsPlayerHost(  
    PEER peer,  
    const gsi_char * nick,  
    RoomType roomType );
```

Routine	Required Header	Distribution
peerIsPlayerHost	<peer.h>	SDKZIP

Return Value

This function returns PEERTrue if the specified player is a room host or operator. PEERFalse otherwise.

Parameters

peer

[in] Initialized peer object.

nick

[in] Nickname of the target player.

roomType

[in] Can be either TitleRoom, GroupRoom or StagingRoom.

Remarks

The **peerIsPlayerHost** function checks whether the player that the nick refers to is the host or operator of the room. It returns `PEERFalse` if that player isn't a host or operator. This simply is a shortcut to obtaining the player's flags, e.g. the flags `PEER_FLAG_HOST` and `PEER_FLAG_OP`. This function will fail if no title is set and the peer object is not connected.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
peerIsPlayerHost	peerIsPlayerHostA	peerIsPlayerHostW

peerIsPlayerHostW and **peerIsPlayerHostA** are UNICODE and ANSI mapped versions of **peerIsPlayerHost**. The arguments of **peerIsPlayerHostA** are ANSI strings; those of **peerIsPlayerHostW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

peerIsPlaying

Returns PEERTrue if the local client is playing.

```
PEERBool peerIsPlaying(  
    PEER peer );
```

Routine	Required Header	Distribution
peerIsPlaying	<peer.h>	SDKZIP

Return Value

This function returns `PEERTrue` if the local client is playing. `PEERFalse` otherwise.

Parameters

peer

[in] Initialized peer object.

Remarks

The **peerIsPlaying** checks to see if the local player is playing. This will be true if the player successfully launched a game with `peerStartGame()`, was in a staging room and got the `peerGameStartedCallback()`, or was marked as playing with `peerStartPlaying()`. This function can only be called if a title is set and the peer object is connected.

Section Reference: [Gamespy Peer SDK](#)

peerJoinGroupRoom

Join the specified group room.

```
void peerJoinGroupRoom(  
    PEER peer,  
    int groupID,  
    peerJoinRoomCallback callback,  
    void * param,  
    PEERBool blocking );
```

Routine	Required Header	Distribution
peerJoinGroupRoom	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object.

groupID

[in] ID number of the group. See remarks.

callback

[in] Callback function to be called when the operation completes.

param

[in] Pointer to user data. This is optional and will be passed unmodified to the callback function.

blocking

[in] When set to PEERTrue this function will not return until the operation has completed.

Remarks

The **peerJoinGroupRoom** function may be used to join the GroupRoom which matches the specified groupID. Group IDs may be obtained through the peerListGroupRooms call. The peer object must be connected to the chat server and have a title set before joining any rooms.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
peerJoinGroupRoom	peerJoinGroupRoomA	peerJoinGroupRoomW

peerJoinGroupRoomW and **peerJoinGroupRoomA** are UNICODE and ANSI mapped versions of **peerJoinGroupRoom**. The arguments of **peerJoinGroupRoomA** are ANSI strings; those of **peerJoinGroupRoomW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

peerJoinStagingRoom

Joins a specified game staging room. Allows players to get together and chat while setting up a game. Players can also see other players' pings and crosspings.

```
void peerJoinStagingRoom(  
    PEER peer,  
    SBServer server,  
    const gsi_char password[PEER_PASSWORD_LEN],  
    peerJoinRoomCallback callback,  
    void * param,  
    PEERBool blocking );
```

Routine	Required Header	Distribution
peerJoinStagingRoom	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object.

server

[in] SBServer object.

password

[in] Optional password for the staging room.

callback

[in] Callback function to be called when the operation completes.

param

[in] Pointer to user data. This is optional and will be passed unmodified to the callback function.

blocking

[in] When set to PEERTrue this function will not return until the operation has completed.

Remarks

The **peerJoinStagingRoom** connects a player to a staging room. "server" is one of the server objects passed to `peerListingGamesCallback()`. "password" will be checked to match the password passed to `peerCreateStaginRoom`. Spaces in passwords are not allowed. Any password with spaces should be stripped of those spaces before calling this function, or a warning to the user will suffice. This call will only work if the server was listed with "staging" set to true. Otherwise the game has already been launched and should be joined directly. As long as the server object was obtained from a `peerStartListGames()` call that was made after the most recent call to `peerSetTitle()`, then the listing can be safely stopped (with `peerStopListingGames()`) before making this call.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
peerJoinStagingRoom	peerJoinStagingRoomA	peerJoinStagingRoomW

peerJoinStagingRoomW and **peerJoinStagingRoomA** are UNICODE and ANSI mapped versions of **peerJoinStagingRoom**. The arguments of **peerJoinStagingRoomA** are ANSI strings; those of **peerJoinStagingRoomW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

See Also: [peerJoinRoomCallback](#)

peerJoinStagingRoomByChannel

Join a staging room using the channel name.

```
void peerJoinStagingRoomByChannel(  
    PEER peer,  
    const gsi_char * channel,  
    const gsi_char password[PEER_PASSWORD_LEN],  
    peerJoinRoomCallback callback,  
    void * param,  
    PEERBool blocking );
```

Routine	Required Header	Distribution
peerJoinStagingRoomByChannel	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object.

channel

[in] Chat channel name.

password

[in] Optional password of the room.

callback

[in] Callback function to be called when the operation completes.

param

[in] Pointer to user data. This is optional and will be passed unmodified to the callback function.

blocking

[in] When set to PEERTrue this function will not return until the operation has completed.

Remarks

The **peerJoinStagingRoomByChannel** connects the local player to a staging room by using its channel name. A "password" will be checked to match the password passed to `peerCreateStagingRoom`. Spaces in passwords are not allowed. Any password with spaces should be stripped of those spaces before calling this function, or a warning to the user will suffice. This call will only work if the server was listed with "staging" set to true. Otherwise the game has already been launched and should be joined directly. The function will fail if the channel does not have "staging" set to true.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI
peerJoinStagingRoomByChannel	peerJoinStagingRoomByChannelA	pee

peerJoinStagingRoomByChannelW and **peerJoinStagingRoomByChannelA** are UNICODE and ANSI mapped versions of **peerJoinStagingRoomByChannel**. The arguments of **peerJoinStagingRoomByChannelA** are ANSI strings; those of **peerJoinStagingRoomByChannelW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

peerJoinTitleRoom

Join the title room for the local client's game application.

```
void peerJoinTitleRoom(  
    PEER peer,  
    const gsi_char password[PEER_PASSWORD_LEN],  
    peerJoinRoomCallback callback,  
    void * param,  
    PEERBool blocking );
```

Routine	Required Header	Distribution
peerJoinTitleRoom	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object.

password

[in] Optional password of the title room. Usually NULL.

callback

[in] Callback function to be called when the operation completes.

param

[in] Pointer to user data. This is optional and will be passed unmodified to the callback function.

blocking

[in] When set to PEERTrue this function will not return until the operation has completed.

Remarks

The **peerJoinTitleRoom** function Joins the title room of the currently selected game title. This is typically called right after connecting, and it is only valid if a title is set. If the user is already in a title room, this function will fail.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
<code>peerJoinTitleRoom</code>	<code>peerJoinTitleRoomA</code>	<code>peerJoinTitleRoomW</code>

peerJoinTitleRoomW and **peerJoinTitleRoomA** are UNICODE and ANSI mapped versions of **peerJoinTitleRoom**. The arguments of **peerJoinTitleRoomA** are ANSI strings; those of **peerJoinTitleRoomW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

peerKickPlayer

Kick a player from a room.

```
void peerKickPlayer(  
    PEER peer,  
    RoomType roomType,  
    const gsi_char * nick,  
    const gsi_char * reason );
```

Routine	Required Header	Distribution
peerKickPlayer	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object.

roomType

[in] Can be either TitleRoom, GroupRoom or StagingRoom.

nick

[in] Nickname of the client to kick.

reason

[in] Optional explanation string to be sent to the target client.

Remarks

The **peerKickPlayer** function will kick a specified player from the room. Only players that have operator or host ability will be able to kick players. A player can be kicked by the operator with or without reason. An player that does not exist cannot be kicked. If the player does not exist in this room, then no one will be kicked.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
peerKickPlayer	peerKickPlayerA	peerKickPlayerW

peerKickPlayerW and **peerKickPlayerA** are UNICODE and ANSI mapped versions of **peerKickPlayer**. The arguments of **peerKickPlayerA** are ANSI strings; those of **peerKickPlayerW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

peerLeaveRoom

Remove the local client from a room.

```
void peerLeaveRoom(  
    PEER peer,  
    RoomType roomType,  
    const gsi_char * reason );
```

Routine	Required Header	Distribution
peerLeaveRoom	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object.

roomType

[in] Can be either TitleRoom, GroupRoom or StagingRoom.

reason

[in] Optional text reason broadcast to the room.

Remarks

The **peerLeaveRoom** function will take the local player out of the current room. Any room can be left without affecting any other room.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
peerLeaveRoom	peerLeaveRoomA	peerLeaveRoomW

peerLeaveRoomW and **peerLeaveRoomA** are UNICODE and ANSI mapped versions of **peerLeaveRoom**. The arguments of **peerLeaveRoomA** are ANSI strings; those of **peerLeaveRoomW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

peerListGroupRooms

List all the group rooms for the currently set title.

```
void peerListGroupRooms(  
    PEER peer,  
    const gsi_char * fields,  
    peerListGroupRoomsCallback callback,  
    void * param,  
    PEERBool blocking );
```

Routine	Required Header	Distribution
peerListGroupRooms	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object.

fields

[in] Backslash delimited list of fields.

callback

[in] Callback function to be called when the operation completes.

param

[in] Pointer to user data. This is optional and will be passed unmodified to the callback function.

blocking

[in] When set to PEERTrue this function will not return until the operation has completed.

Remarks

The **peerListGroupRooms** function lists all the groups rooms for the currently set title. The callback will be called once for each title, then it will be called once again with a "groupID" of 0. The groupIDs can be used with `peerJoinGroupRoom()` to join a group. This function will fail if no title is set.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
peerListGroupRooms	peerListGroupRoomsA	peerListGroupRoomsW

peerListGroupRoomsW and **peerListGroupRoomsA** are UNICODE and ANSI mapped versions of **peerListGroupRooms**. The arguments of **peerListGroupRoomsA** are ANSI strings; those of **peerListGroupRoomsW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

peerMessagePlayer

Send a message to the specified player.

```
void peerMessagePlayer(  
    PEER peer,  
    const gsi_char * nick,  
    const gsi_char * message,  
    MessageType messageType );
```

Routine	Required Header	Distribution
peerMessagePlayer	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object.

nick

[in] Nickname of the player who should receive the message.

message

[in] The message to send.

messageType

[in] The type of message to send, most commonly NormalMessage or ActionMessage.

Remarks

The **peerMessagePlayer** function sends a message to another player. As long as the nick is valid and matches up with a player, that player will get the message in a `peerPlayerMessageCallback()`. This function only works while connected to the chat server. The player messaged does not have to be using the Peer SDK.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
peerMessagePlayer	peerMessagePlayerA	peerMessagePlayerW

peerMessagePlayerW and **peerMessagePlayerA** are UNICODE and ANSI mapped versions of **peerMessagePlayer**. The arguments of **peerMessagePlayerA** are ANSI strings; those of **peerMessagePlayerW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

peerMessageRoom

Send a message to the specified room.

```
void peerMessageRoom(  
    PEER peer,  
    RoomType roomType,  
    const gsi_char * message,  
    MessageType messageType );
```

Routine	Required Header	Distribution
peerMessageRoom	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object.

roomType

[in] Can be either TitleRoom, GroupRoom or StagingRoom.

message

[in] The message to send.

messageType

[in] The type of message to send, most commonly NormalMessage or ActionMessage.

Remarks

The **peerMessageRoom** function sends a message to a room. All the players in the room, including the local player, will receive a `peerRoomMessageCallback()` with the message. This function only works if the local user is in the room he is trying to message. The peer object must be connected and a title must be set for this function to successfully work.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
peerMessageRoom	peerMessageRoomA	peerMessageRoomW

peerMessageRoomW and **peerMessageRoomA** are UNICODE and ANSI mapped versions of **peerMessageRoom**. The arguments of **peerMessageRoomA** are ANSI strings; those of **peerMessageRoomW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

peerParseQuery

Pass a manually received server query to the peer SDK. Use with `peerStartReportingWithSocket` or `peerCreateStagingRoomWithSocket`.

```
void peerParseQuery(  
    PEER peer,  
    gsi_char * query,  
    int len,  
    struct sockaddr * sender );
```

Routine	Required Header	Distribution
peerParseQuery	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object.

query

[in] String of query data received on the socket.

len

[in] String length of query, not including the NULL.

sender

[in] The address this query was received from.

Remarks

The **peerParseQuery** can be used to translate peer packets into data. If hosting a room or a game using shared sockets, then this function needs to be used to pass any data received on that socket to Peer. When data is received on the socket, the application must determine if the data is a query meant to be passed to Peer, or if it is data for the game itself. This can be done by checking the first two bytes in a packet for QR_MAGIC_1 and QR_MAGIC_2.

Again, this function only needs to be called if reporting over a shared socket, which is initiated by either `peerCreateStagingRoomWithSocket()` or `peerStartReportingWithSocket()`.

Section Reference: [Gamespy Peer SDK](#)

peerPingPlayer

Send a ping request to a remote player.

```
PEERBool peerPingPlayer(  
    PEER peer,  
    const gsi_char * nick );
```

Routine	Required Header	Distribution
peerPingPlayer	<peer.h>	SDKZIP

Return Value

Returns PEERTrue if a ping attempt is made, PEERFalse otherwise.

Parameters

peer

[in] Initialized peer object.

nick

[in] Nickname of the remote player to ping.

Remarks

The **peerPingPlayer** function sends a UDP ping to the specified player. Peer already automatically pings all players that are in ping rooms (which are set in `peerSetTitle`). This function does a one-time ping of a remote player in a non-ping room. However pings must be enabled in at least one room for this to work, otherwise Peer will not open the UDP ping socket. Also, `peerAlwaysGetPlayerInfo()` must be enabled so that Peer has IPs for players that are only in non-ping rooms.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
peerPingPlayer	peerPingPlayerA	peerPingPlayerW

peerPingPlayerW and **peerPingPlayerA** are UNICODE and ANSI mapped versions of **peerPingPlayer**. The arguments of **peerPingPlayerA** are ANSI strings; those of **peerPingPlayerW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

peerRegisterUniqueNick

Register a unique nick. Call in response to peerNickErrorCallback.

```
void peerRegisterUniqueNick(  
    PEER peer,  
    int namespaceID,  
    const gsi_char * uniquenick,  
    const gsi_char * cdkey );
```

Routine	Required Header	Distribution
peerRegisterUniqueNick	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object.

namespaceID

[in] Assigned namespace ID for this title.

uniquenick

[in] Unique nickname to register with cdkey.

cdkey

[in] User's cdkey. Uniquenick will be attached to this key.

Remarks

The **peerRegisterUniqueNick** takes and registers a unique nick supplied. If the `nickErrorCallback` was called with a type of `PEER_UNIQUE_NICK_EXPIRED` or `PEER_NO_UNIQUE_NICK`, then this function can be called to associate a `uniquenick` with the profile which is being used to login (passed to `peerConnectLogin`). If `uniquenick` is `NULL` or an empty string, then the connect attempt will be aborted. This function should only be called in response to a `peerNickErrorCallback()`. The backend makes certain checks on a `uniquenick` before it is allowed to be registered. For details on what is checked, see the "Unique Nick Checks" section at the bottom of GameSpy Presence SDK overview documentation.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defin
peerRegisterUniqueNick	peerRegisterUniqueNickA	peerRegisterUniqueN

peerRegisterUniqueNickW and **peerRegisterUniqueNickA** are UNICODE and ANSI mapped versions of **peerRegisterUniqueNick**. The arguments of **peerRegisterUniqueNickA** are ANSI strings; those of **peerRegisterUniqueNickW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

peerRetryWithNick

Use in response to a `nickErrorCallback`. This function allows the local client to retry the connection attempt with a different chat nickname.

```
void peerRetryWithNick(  
    PEER peer,  
    const gsi_char * nick );
```

Routine	Required Header	Distribution
<code>peerRetryWithNick</code>	<code><peer.h></code>	SDKZIP

Parameters

peer

[in] Initialized peer object.

nick

[in] Set to zero unless directed otherwise by GameSpy.

Remarks

The **peerRetryWithNick** function should be used in response to a `nickErrorCallback`. Most often, this occurs when a requested nickname is already in use. **peerRetryWithNick** should be called with an alternate nickname such as "oldnick{1}" to continue the login process. If another `nickError` occurs, the `nickErrorCallback` will be triggered again.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
peerRetryWithNick	peerRetryWithNickA	peerRetryWithNickW

peerRetryWithNickW and **peerRetryWithNickA** are UNICODE and ANSI mapped versions of **peerRetryWithNick**. The arguments of **peerRetryWithNickA** are ANSI strings; those of **peerRetryWithNickW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

peerSendNatNegotiateCookie

Send a nat negotiation cookie to a server.

```
void peerSendNatNegotiateCookie(  
    PEER peer,  
    unsigned int ip,  
    unsigned short port,  
    int cookie );
```

Routine	Required Header	Distribution
peerSendNatNegotiateCookie	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object.

ip

[in] IP address of the remote server.

port

[in] Port of the remote server.

cookie

[in] Cookie to send. Usually an integer value randomly generated by the sender.

Remarks

The `peerSendNatNegotiationCookie` will send a nat-negotiate cookie to the master server. See the nat-negotiate documentation for more information.

Section Reference: [Gamespy Peer SDK](#)

peerSetAwayMode

Set the away mode, as it appears in chat.

```
void peerSetAwayMode(  
    PEER peer,  
    const gsi_char * reason );
```

Routine	Required Header	Distribution
peerSetAwayMode	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object.

reason

[in] Reason for being away. Set to NULL or "" if not away.

Remarks

The **peerSetAwayMode** sets the away mode of the local player. If an empty string or NULL, away mode is off. If a valid string, away mode is on. Once on, away mode will stay active until it is turned off, or the connection is disconnected. To check if a player is away, check his flags (with `peerGetPlayerFlags()`), and check for `PEER_FLAG_AWAY`. To get the reason, check for that player's special "away" global key (with `peerGetPlayerGlobalKeys()`). This function has no effect if peer object is not connected.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
peerSetAwayMode	peerSetAwayModeA	peerSetAwayModeW

peerSetAwayModeW and **peerSetAwayModeA** are UNICODE and ANSI mapped versions of **peerSetAwayMode**. The arguments of **peerSetAwayModeA** are ANSI strings; those of **peerSetAwayModeW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

peerSetGlobalKeys

Set global keys on the local player.

```
void peerSetGlobalKeys(  
    PEER peer,  
    int num,  
    const gsi_char ** keys,  
    const gsi_char ** values );
```

Routine	Required Header	Distribution
peerSetGlobalKeys	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object.

num

[in] Number of keys in the keys array.

keys

[in] Array of keys to set values for.

values

[in] Array of values to set.

Remarks

The **peerSetGlobalKeys** function sets global keys on the local player. At least one key must be set. This will fail if the peer object is not connected.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
<code>peerSetGlobalKeys</code>	<code>peerSetGlobalKeysA</code>	<code>peerSetGlobalKeysW</code>

peerSetGlobalKeysW and **peerSetGlobalKeysA** are UNICODE and ANSI mapped versions of **peerSetGlobalKeys**. The arguments of **peerSetGlobalKeysA** are ANSI strings; those of **peerSetGlobalKeysW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

peerSetGlobalWatchKeys

Set the global watch keys for the specified room type.

```
void peerSetGlobalWatchKeys(  
    PEER peer,  
    RoomType roomType,  
    int num,  
    const gsi_char ** keys,  
    PEERBool addKeys );
```

Routine	Required Header	Distribution
peerSetGlobalWatchKeys	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object.

roomType

[in] Can be either TitleRoom, GroupRoom or StagingRoom.

num

[in] Number of keys in the array.

keys

[in] Array of keys to watch for.

addKeys

[in] When set to PEERTrue this keys will be added to the existing watch key list.

Remarks

The **peerSetGlobalWatchKeys** sets the global watch keys for a room type. If `addKeys` is set to `PEERTrue`, the keys will be added to the current global watch keys for this room. If `addKeys` is `PEERFalse`, these will replace any existing global watch keys for this room. When entering a room of the given type, `peer` will get and cache these keys for all players in the room. To check the value of a key at any time, use `peerGetGlobalWatchKey()`. This will fail if no title is set and if `peer` is not connected.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE De
peerSetGlobalWatchKeys	peerSetGlobalWatchKeysA	peerSetGlobalWat

peerSetGlobalWatchKeysW and **peerSetGlobalWatchKeysA** are UNICODE and ANSI mapped versions of **peerSetGlobalWatchKeys**. The arguments of **peerSetGlobalWatchKeysA** are ANSI strings; those of **peerSetGlobalWatchKeysW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

peerSetGroupID

Manually set the group ID. Use with caution as this is normally set automatically.

```
void peerSetGroupID(  
    PEER peer,  
    int groupID );
```

Routine	Required Header	Distribution
peerSetGroupID	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object.

groupID

[in] Integer ID to set. "0" for no group.

Remarks

The **peerSetGroupID** sets the group ID of a group room manually. This is not safe for automatically assigned group IDs.

Section Reference: [Gamespy Peer SDK](#)

peerSetPassword

Set the password on the chat room. Local client must be the host.

```
void peerSetPassword(  
    PEER peer,  
    RoomType roomType,  
    const gsi_char password[PEER_PASSWORD_LEN] );
```

Routine	Required Header	Distribution
peerSetPassword	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object.

roomType

[in] Can be either TitleRoom, GroupRoom or StagingRoom.

password

[in] Password to set on this room.

Remarks

The **peerSetPassword** sets a password in a room a local player hosting. The only RoomType currently supported is StagingRoom. This will only work if the player is hosting the room. If password is NULL or "", the password will be cleared.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
peerSetPassword	peerSetPasswordA	peerSetPasswordW

peerSetPasswordW and **peerSetPasswordA** are UNICODE and ANSI mapped versions of **peerSetPassword**. The arguments of **peerSetPasswordA** are ANSI strings; those of **peerSetPasswordW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

peerSetQuietMode

Sets the peer sdk to quiet mode or disables quiet mode. See remarks.

```
void peerSetQuietMode(  
    PEER peer,  
    PEERBool quiet );
```

Routine	Required Header	Distribution
peerSetQuietMode	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object.

quiet

[in] Set to PEERTrue to enable quiet mode. PEERFalse to disable.

Remarks

The **peerSetQuietMode** function is used to toggle quiet mode. When in quiet mode the peer SDK will not receive chat or other messages. This allows the user to remain logged into chat without disrupting gameplay with extraneous traffic. If quiet mode is enabled, the chat server will not send this user channel messages or channel join/parts. This will last until quiet mode is disabled. At that time the `peerNewPlayerListCallback()` will be called for each room the local user is in with the current list of players. This function has no effect if the title is not set and the peer object is not connected.

Section Reference: [Gamespy Peer SDK](#)

peerSetReady

Set the local clients ready state.

```
void peerSetReady(  
    PEER peer,  
    PEERBool ready );
```

Routine	Required Header	Distribution
peerSetReady	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object.

ready

[in] Set to PEERTrue to enable ready status.

Remarks

The **peerSetReady** function allows the local client to signal his/her ready status for a staging room. A staging room host should not launch the game until all clients are ready.

Section Reference: [Gamespy Peer SDK](#)

peerSetRoomKeys

Set room keys for a player or the room itself.

```
void peerSetRoomKeys(  
    PEER peer,  
    RoomType roomType,  
    const gsi_char * nick,  
    int num,  
    const gsi_char ** keys,  
    const gsi_char ** values );
```

Routine	Required Header	Distribution
peerSetRoomKeys	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object.

roomType

[in] Can be either TitleRoom, GroupRoom or StagingRoom.

nick

[in] Nickname of the target player.

num

[in] Number of keys to set.

keys

[in] Array of keys to set values for.

values

[in] Array of values to set.

Remarks

The **peerSetRoomKeys** function is used to set a player's room keys or to set a room's keys. Only hosts can set keys on the room, and only hosts can set keys on other players. This will fail if no title is set and the peer is not connected.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
<code>peerSetRoomKeys</code>	<code>peerSetRoomKeysA</code>	<code>peerSetRoomKeysW</code>

peerSetRoomKeysW and **peerSetRoomKeysA** are UNICODE and ANSI mapped versions of **peerSetRoomKeys**. The arguments of **peerSetRoomKeysA** are ANSI strings; those of **peerSetRoomKeysW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

peerSetRoomName

Set the name of a room. Local client must be the host.

```
void peerSetRoomName(  
    PEER peer,  
    RoomType roomType,  
    const gsi_char * name );
```

Routine	Required Header	Distribution
peerSetRoomName	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object.

roomType

[in] Can be either TitleRoom, GroupRoom or StagingRoom.

name

[in] The new name.

Remarks

The **peerSetRoomName** function sets the name of a room you're hosting. The only RoomType currently supported is StagingRoom. This will only work if the player is hosting the room.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
peerSetRoomName	peerSetRoomNameA	peerSetRoomNameW

peerSetRoomNameW and **peerSetRoomNameA** are UNICODE and ANSI mapped versions of **peerSetRoomName**. The arguments of **peerSetRoomNameA** are ANSI strings; those of **peerSetRoomNameW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

peerSetStagingRoomMaxPlayers

Update the maximum number of players for a staging room. Local client must be the host.

```
void peerSetStagingRoomMaxPlayers(  
    PEER peer,  
    int maxPlayers );
```

Routine	Required Header	Distribution
peerSetStagingRoomMaxPlayers	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object.

maxPlayers

[in] Maximum number of players.

Remarks

The **peerSetStagingRoomMaxPlayers** function updates the maximum number of players that a staging room can have. This can only be set by the host of the staging room.

Section Reference: [Gamespy Peer SDK](#)

peerSetTitle

Set the game information to be used by the peer sdk.

```
PEERBool peerSetTitle(  
    PEER peer,  
    const gsi_char * title,  
    const gsi_char * qrSecretKey,  
    const gsi_char * sbName,  
    const gsi_char * sbSecretKey,  
    int sbGameVersion,  
    int sbMaxUpdates,  
    PEERBool natNegotiate,  
    PEERBool pingRooms[NumRooms],  
    PEERBool crossPingRooms[NumRooms] );
```

Routine	Required Header	Distribution
peerSetTitle	<peer.h>	SDKZIP

Return Value

This function returns `PEERFalse` if an error occurs, otherwise `PEERTrue`.

Parameters

peer

[in] Initialized peer object.

title

[in] Your gamename, assigned by GameSpy.

qrSecretKey

[in] Secret key for title, assigned by GameSpy.

sbName

[in] Servers returned will be for this GameName. (Usually the same as title.)

sbSecretKey

[in] Secret key for sbName.

sbGameVersion

[in] Game version for the local client.

sbMaxUpdates

[in] The maximum number of server queries the SDK will send out at one time.

natNegotiate

[in] Set to PEERTrue if nat negotiation is supported.

pingRooms

[in] Array of PEERBool, use to indicate which room types to automatically ping.

crossPingRooms

[in] Array of PEERBool, use to indicate which room types to automatically cross ping.

Remarks

The **peerSetTitle** function sets title information for the peer sdk such as that used in server browsing and reporting. This should be called after `peerInitialize` and before any of the connection functions: `peerConnect`, `peerConnectLogin`, `peerConnectPreAuth`.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
peerSetTitle	peerSetTitleA	peerSetTitleW

peerSetTitleW and **peerSetTitleA** are UNICODE and ANSI mapped versions of **peerSetTitle**. The arguments of **peerSetTitleA** are ANSI strings; those of **peerSetTitleW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

peerSetTitleRoomChannel

Set the channel to be used as the TitleRoom. (Rarely used, SDK sets title room automatically.).

```
void peerSetTitleRoomChannel(  
    PEER peer,  
    const gsi_char * channel );
```

Routine	Required Header	Distribution
peerSetTitleRoomChannel	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object.

channel

[in] Name of the channel to use as the title room.

Remarks

The **peerSetTitleRoomChannel** function associates a channel to be used as the TitleRoom. This function is normally not needed. It must be called while a title is set, and will only last until a new title is set. If called with a NULL or empty channel, then peer will determine the channel (the default behavior). If this is called while in a title room, it won't take effect until the title room is left.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE D
peerSetTitleRoomChannel	peerSetTitleRoomChannelA	peerSetTitleRoom

peerSetTitleRoomChannelW and **peerSetTitleRoomChannelA** are UNICODE and ANSI mapped versions of **peerSetTitleRoomChannel**. The arguments of **peerSetTitleRoomChannelA** are ANSI strings; those of **peerSetTitleRoomChannelW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

peerShutdown

Destructs the peer SDK.

```
void peerShutdown(  
    PEER peer );
```

Routine	Required Header	Distribution
peerShutdown	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object.

Remarks

The **peerShutdown** function is used to destruct the peer sdk. This frees all internally allocated memory. The peer sdk object should not be used after this call. To use the SDK again, call `peerInitialize()`. This call will also disconnect any outstanding connections.

Section Reference: [Gamespy Peer SDK](#)

peerStartAutoMatch

Start an automatch attempt.

```
void peerStartAutoMatch(  
    PEER peer,  
    int maxPlayers,  
    const gsi_char * filter,  
    peerAutoMatchStatusCallback statusCallback,  
    peerAutoMatchRateCallback rateCallback,  
    void * param,  
    PEERBool blocking );
```

Routine	Required Header	Distribution
peerStartAutoMatch	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object.

maxPlayers

[in] Total number of players to match. (Includes local player).

filter

[in] Hard filter for returned server list.

statusCallback

[in] Callback function to be called when the status changes.

rateCallback

[in] Callback function to be called when a potential match should be rated.

param

[in] Pointer to user data. This is optional and will be passed unmodified to the callback function.

blocking

[in] When set to PEERTrue this function will not return until the operation has completed.

Remarks

The **peerStartAutoMatch** function begins the AutoMatch process. The function takes a `maxPlayers` parameter, which specifies the maximum number of people that should be in the final match. For example, if the local player wants to play a 3v3 match, `maxPlayers` should be 6. The match can be started before the `maxPlayers` is reached, for example if an exact number of players is not needed, but an upper limit must still be specified.

The filter is a SQL-type filter, just like the filter used in `peerStartListingGames()`. It is used to rule out matches that are not acceptable. Potential matches that do not pass the filter will not be passed to the rating callback. The `statusCallback` is called whenever the status of the match changes, until either `peerStopAutoMatch()` is called, or the `statusCallback` is called with a status of `PEERFailed` or `PEERComplete`. The `rateCallback` is used to rate possible matches.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
peerStartAutoMatch	peerStartAutoMatchA	peerStartAutoMatchW

peerStartAutoMatchW and **peerStartAutoMatchA** are UNICODE and ANSI mapped versions of **peerStartAutoMatch**. The arguments of **peerStartAutoMatchA** are ANSI strings; those of **peerStartAutoMatchW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

peerStartAutoMatchWithSocket

Start an automatch attempt using an external managed socket.

```
void peerStartAutoMatchWithSocket(  
    PEER peer,  
    int maxPlayers,  
    const gsi_char * filter,  
    SOCKET socket,  
    unsigned short port,  
    peerAutoMatchStatusCallback statusCallback,  
    peerAutoMatchRateCallback rateCallback,  
    void * param,  
    PEERBool blocking );
```

Routine	Required Header	Distribution
peerStartAutoMatchWithSocket	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object.

maxPlayers

[in] Total number of players to match. (Includes local player).

filter

[in] Hard filter for returned server list.

socket

[in] Socket to be used for reporting.

port

[in] Local port to which the socket is bound.

statusCallback

[in] Callback function to be called when the status changes.

rateCallback

[in] Callback function to be called when a potential match should be rated.

param

[in] Pointer to user data. This is optional and will be passed unmodified to the callback function.

blocking

[in] When set to PEERTrue this function will not return until the operation has completed.

Remarks

The `peerStartAutoMatch` function begins the `AutoMatch` process using the specified socket being shared. The function takes a `maxPlayers` parameter, which specifies the maximum number of people that should be in the final match. For example, if the local player wants to play a 3v3 match, `maxPlayers` should be 6. The match can be started before the `maxPlayers` is reached, for example if an exact number of players is not needed, but an upper limit must still be specified.

The filter is a SQL-type filter, just like the filter used in `peerStartListingGames()`. It is used to rule out matches that are not acceptable. Potential matches that do not pass the filter will not be passed to the rating callback. The `statusCallback` is called whenever the status of the match changes, until either `peerStopAutoMatch()` is called, or the `statusCallback` is called with a status of `PEERFailed` or `PEERComplete`. The `rateCallback` is used to rate possible matches.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UN
peerStartAutoMatchWithSocket	peerStartAutoMatchWithSocketA	peerSta

peerStartAutoMatchWithSocketW and **peerStartAutoMatchWithSocketA** are UNICODE and ANSI mapped versions of **peerStartAutoMatchWithSocket**. The arguments of **peerStartAutoMatchWithSocketA** are ANSI strings; those of **peerStartAutoMatchWithSocketW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

peerStartGame

Called by the host to begin the game.

```
void peerStartGame(  
    PEER peer,  
    const gsi_char * message,  
    int reportingOptions );
```

Routine	Required Header	Distribution
peerStartGame	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object.

message

[in] Optional message to send to each client.

reportingOptions

[in] Bitfield flags used to set reporting options. (example:
PEER_KEEP_REPORTING)

Remarks

The **peerStartGame** function is called only by a staging room host to start the game. All the other people in the staging room will have their `peerGameStartedCallback()` called. The message gets passed to everyone in the callback, and can be used to pass information such as a special port or password.

Peer does not enforce readiness -- this function can be called at any time, no matter who is ready or not. If the application wishes to only launch the game once everyone is ready, then it is up to the application to enforce that.

If `PEER_STOP_REPORTING` is set in `reportingOptions`, Peer will stop server reporting, and the program is responsible from then on for reporting the server to the backend. If `PEER_KEEP_REPORTING` is set instead, Peer will continue doing server reporting, and calling the program-supplied callbacks. Peer will normally not report all the same information while playing. While playing, the application will be responsible for reporting the gamemode (if not openplaying), the hostname, the numplayers, the maxplayers, and any password in the callbacks, unless the `PEER_REPORT_INFO` flag is set in `reportingOptions`. The application will also need to report the players and pings in the callbacks, unless the `PEER_REPORT_PLAYERS` flag is set in `reportingOptions`. For more details, see the `peerQR*Callback()` in the callback section below.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
peerStartGame	peerStartGameA	peerStartGameW

peerStartGameW and **peerStartGameA** are UNICODE and ANSI mapped versions of **peerStartGame**. The arguments of **peerStartGameA** are ANSI strings; those of **peerStartGameW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

peerStartListingGames

Begin listing the currently running games and staging rooms.

```
void peerStartListingGames(  
    PEER peer,  
    const unsigned char * fields,  
    int numFields,  
    const gsi_char * filter,  
    peerListingGamesCallback callback,  
    void * param );
```

Routine	Required Header	Distribution
peerStartListingGames	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object.

fields

[in] Array of registered QR2 keys to request from the servers.

numFields

[in] Number of keys in the fields array.

filter

[in] SQL-link rule filter.

callback

[in] Callback function to be called when each server updates.

param

[in] Pointer to user data. This is optional and will be passed unmodified to the callback function.

Remarks

The **peerStartListingGames** function starts listing the currently running games and staging rooms. This is used to maintain a list that can be presented to the user, so they can pick a game (or staging room) to join. Games and staging rooms are filtered based on what group the local user is in. If the local user isn't in a group, then only games and staging rooms that aren't part of any group are listed. Peer first gets an initial list of existing servers, then it receives continuous updates about new servers, deleted servers, and updated servers. The callback will keep being called repeatedly with new information until the listing is stopped with `peerStopListingGames()`, or the title is cleared or changed with `peerClearTitle()` or `peerSetTitle()`.

The filter can be a SQL-style Boolean statement, such as:

`"gametype='ctf'"` or

`"numplayers > 1 and numplayers < 8"`.

The filter can be arbitrarily complex and supports all standard SQL groupings and Boolean operations. The following fields are available for filtering: `hostport`, `gamever`, `location`, `hostname`, `mapname`, `gametype`, `gamemode`, `numplayers`, `maxplayers`, `groupid`.

This function will fail if there is no title set.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
peerStartListingGames	peerStartListingGamesA	peerStartListingGamesV

peerStartListingGamesW and **peerStartListingGamesA** are UNICODE and ANSI mapped versions of **peerStartListingGames**. The arguments of **peerStartListingGamesA** are ANSI strings; those of **peerStartListingGamesW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

peerStartPlaying

Flag the local player as "playing". Use this to manual set the player's flag in the event a non peer sdk game is started.

```
void peerStartPlaying(  
    PEER peer );
```

Routine	Required Header	Distribution
peerStartPlaying	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object.

Remarks

The **peerStartPlaying** function is used to manually set the local clients "playing" flag. This notifies other clients that the local client has entered a game. This is only necessary when the peer sdk is unable to automatically detect the state change, as it does with the "peerStartGame" call.

Section Reference: [Gamespy Peer SDK](#)

peerStartReporting

Begins server reporting, does not create a staging room.

```
PEERBool peerStartReporting(  
    PEER peer );
```

Routine	Required Header	Distribution
peerStartReporting	<peer.h>	SDKZIP

Return Value

None.

Parameters

peer

[in] Initialized peer object.

Remarks

The **peerStartReporting** function starts server reporting, without ever creating a staging room. This would be used if the application needs to start reporting itself as a server without any staging room attached. Call `peerStopGame()` when done to stop reporting.

Section Reference: [Gamespy Peer SDK](#)

peerStartReportingWithSocket

Begin server reporting using an externally managed socket.

```
PEERBool peerStartReportingWithSocket(  
    PEER peer,  
    SOCKET socket,  
    unsigned short port );
```

Routine	Required Header	Distribution
peerStartReportingWithSocket	<peer.h>	SDKZIP

Return Value

Returns PEERTrue if reporting was successfully started.

Parameters

peer

[in] Initialized peer object.

socket

[in] SOCKET to be used for reporting.

port

[in] Local port that the socket is bound to.

Remarks

The `peerStartReporting` function starts server reporting, without ever creating a staging room. This would be used if the application needs to start reporting itself as a server without any staging room attached. Call `peerStopGame()` when done to stop reporting.

If **`peerStartReportingWithSocket`** is used, the socket provided must be an already created UDP socket. It will be used for sending out query replies, and any queries the application reads off of the socket must be passed to Peer using `peerParseQuery()`. This can be useful when running a game host behind a NAT proxy -- for a full explanation of how this helps, see the "NAT and Firewall Support" appendix in the "GameSpy Query and Reporting 2 SDK" documentation.

A title must be set for this function to succeed, but Peer does not need to be connected. Also, this function cannot be called while already in a staging room.

Section Reference: [Gamespy Peer SDK](#)

peerStateChanged

Notify the backend of a server state change, such as the server becoming full.

```
void peerStateChanged(  
    PEER peer );
```

Routine	Required Header	Distribution
peerStateChanged	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object.

Remarks

The **peerStateChanged** function is called to force peer to query the game again. Use this when you want to make sure that the latest info on the game is available, such as when the level or gametype changes. This should only be called while a game is being reported (either from hosting a staging room or game that peer is reporting, or if `peerStartReporting[WithSocket]()` was called).

Section Reference: [Gamespy Peer SDK](#)

peerStayInRoom

Allows SDK to remain in the title room after peerClearTitle. (Rarely used.).

```
void peerStayInRoom(  
    PEER peer,  
    RoomType roomType );
```

Routine	Required Header	Distribution
peerStayInRoom	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object.

roomType

[in] Only TitleRoom is currently supported.

Remarks

Calling this function signals Peer to stay in a room even if the title is cleared or changed (with `peerClearTitle()` or `peerSetTitle()`). This will only be in effect until the next call to `peerSetTitle()`. Only TitleRoom is currently supported. The function has no effect if no title is set.

Section Reference: [Gamespy Peer SDK](#)

peerStopAutoMatch

Stop an automatch attempt in progress.

```
void peerStopAutoMatch(  
    PEER peer );
```

Routine	Required Header	Distribution
peerStopAutoMatch	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object.

Remarks

The **peerStopAutoMatch** function may be used to cancel an automatch attempt in progress. This is generally used so that a user may cancel the AutoMatch process if a suitable match has not been found. A user may encounter this by having a very narrow search criteria.

Section Reference: [Gamespy Peer SDK](#)

peerStopGame

Called by the host when the game has ended.

```
void peerStopGame(  
    PEER peer );
```

Routine	Required Header	Distribution
peerStopGame	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object.

Remarks

The **peerStopGame** function is called by the host to stop a game. This also makes sure the player is no longer marked as playing. Also, this does any necessary cleanup if the local player was the host. This should be called whenever coming back from a game.

Section Reference: [Gamespy Peer SDK](#)

peerStopListingGames

Stops a server list update in progress. Also stops listening for game state changed messages.

```
void peerStopListingGames(  
    PEER peer );
```

Routine	Required Header	Distribution
peerStopListingGames	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object.

Remarks

The **peerStopListingGames** function will stop a server list update in progress. It will also cause the SDK to stop listening for game state changes. The current server list is NOT cleared and remains accessible. Each game in this server list still is considered a valid game. The only time the games are invalidated or updated is if there is a call to **peerStartListingGames** or the title is cleared.

Section Reference: [Gamespy Peer SDK](#)

peerThink

Allow the Peer SDK to continue processing. Callbacks will be triggered during this call.

```
void peerThink(  
    PEER peer );
```

Routine	Required Header	Distribution
peerThink	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object.

Remarks

The **peerThink** function allows the Peer SDK to continue processing. All network communications, callbacks and other events will happen only during this call. The frequency with which this method is called will affect general performance on the SDK.

Section Reference: [Gamespy Peer SDK](#)

peerTranslateNick

Removes the namespace extension from a nickname. Use this when working with unique nicknames in a public chat room.

```
const gsi_char * peerTranslateNick(  
    gsi_char * nick,  
    const gsi_char * extension );
```

Routine	Required Header	Distribution
peerTranslateNick	<peer.h>	SDKZIP

Return Value

Returns the nickname, stripped of the namespace identifier.

Parameters

nick

[in] The current nickname.

extension

[in] The game extension, assigned by GameSpy. This will be removed from the nickname.

Remarks

The **peerTranslateNick** function is used to remove a namespace extension from a nickname. Nicknames that are registered in a game's namespace will include an indentifying extension, such as "-gspy". This extension should not be displayed to the user, but should be stripped before display.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
<code>peerTranslateNick</code>	<code>peerTranslateNickA</code>	<code>peerTranslateNickW</code>

peerTranslateNickW and **peerTranslateNickA** are UNICODE and ANSI mapped versions of **peerTranslateNick**. The arguments of **peerTranslateNickA** are ANSI strings; those of **peerTranslateNickW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

peerUpdateGame

Send an update query to the specified server.

```
void peerUpdateGame(  
    PEER peer,  
    SBServer server,  
    PEERBool fullUpdate );
```

Routine	Required Header	Distribution
peerUpdateGame	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object.

server

[in] Server to update.

fullUpdate

[in] Set to PEERTrue to retrieve values for all keys.

Remarks

The **peerUpdateGame** function is used to send a query to the specified server. This query will retrieve key values and is used when viewing the server list. This function will obtain the full keys for the specified server.

Section Reference: [Gamespy Peer SDK](#)

peerUpdateGameByMaster

This function updates a server via the master server. Passing in true for fullUpdate will obtain the full keys for that server, otherwise it will only obtain the basic keys.

```
void peerUpdateGameByMaster(  
    PEER peer,  
    SBServer server,  
    fullUpdate PEERBool );
```

Routine	Required Header	Distribution
peerUpdateGameByMaster	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object

server

[in] Server to update

PEERBool

[in] PEERTrue for all server keys, PEERFalse for basic keys

Remarks

This function requires the SDK to have a title set. The function is usually called when the initial server list is complete. Should only be called when updating a single server at a time.

Section Reference: [Gamespy Peer SDK](#)

See Also: [peerUpdateGame](#)

peerUTMPlayer

Send a UTM message to the specified client.

```
void peerUTMPlayer(  
    PEER peer,  
    const gsi_char * nick,  
    const gsi_char * command,  
    const gsi_char * parameters,  
    PEERBool authenticate );
```

Routine	Required Header	Distribution
peerUTMPlayer	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object.

nick

[in] Chat Nickname of the target player.

command

[in] The raw command to send.

parameters

[in] Parameters to send along with the command.

authenticate

[in] Set to PEERTrue to have server authenticate this UTM. Normally set this to PEERFalse.

Remarks

The `peerUTMRoom` function may be used to send a UTM message to another player of the specified room.

As long as the nick is valid and matches up with a player, that player will get the message in a `peerPlayerUTMCallback()`. UTM's are used to pass around arbitrary information between players. The command is a short string identifying this UTM (e.g., RQI, NFO, GML).

This function only works while connected.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
peerUTMPlayer	peerUTMPlayerA	peerUTMPlayerW

peerUTMPlayerW and **peerUTMPlayerA** are UNICODE and ANSI mapped versions of **peerUTMPlayer**. The arguments of **peerUTMPlayerA** are ANSI strings; those of **peerUTMPlayerW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

See Also: [peerUTMRoom](#)

peerUTMRoom

Send a UTM message to each client in the room.

```
void peerUTMRoom(  
    PEER peer,  
    RoomType roomType,  
    const gsi_char * command,  
    const gsi_char * parameters,  
    PEERBool authenticate );
```

Routine	Required Header	Distribution
peerUTMRoom	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object.

roomType

[in] Can be either TitleRoom, GroupRoom or StagingRoom.

command

[in] The raw command to send.

parameters

[in] Parameters to send along with the command.

authenticate

[in] Set to PEERTrue to have server authenticate this UTM.
Normally set this to PEERFalse.

Remarks

The `peerUTMPlayer` function may be used to send a UTM message to the specified client. All the players in the room, including the local player, will receive a `peerRoomUTMCallback` with the UTM. UTM's are used to pass around arbitrary information between players. The command is a short string identifying this UTM (e.g., RQI, NFO, GML). This function only works if the local user is in the room he is trying to message.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
peerUTMRoom	peerUTMRoomA	peerUTMRoomW

peerUTMRoomW and **peerUTMRoomA** are UNICODE and ANSI mapped versions of **peerUTMRoom**. The arguments of **peerUTMRoomA** are ANSI strings; those of **peerUTMRoomW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

See Also: [peerUTMPlayer](#)

Peer SDK Callbacks

peerAuthenticateCDKeyCallback	Called when peerAuthenticateCDKey and attempt to authenticate the CD-Key is finished.
peerAutoMatchRateCallback	Called when rating the server for determining the best match
peerAutoMatchStatusCallback	Called when the Automatch state has changed
peerChangeNickCallback	Callback called after peerChangeNick attempt is finished.
peerConnectCallback	Callback for peerConnect
peerCrossPingCallback	Callback for updated cross-ping between two players in the staging room.
peerDisconnectedCallback	Called when a local player has been disconnected from the chat server for any reason
peerEnumPlayersCallback	Called for peerEnumPlayers for each player
peerGameStartedCallback	Called when the host of a staging room launches the game.

peerGetGlobalKeysCallback	Callback for peerGetPlayerGlobalKeys() and peerGetRoomGlobalKeys().
peerGetPlayerInfoCallback	Called after an attempt to peerGetPlayerInfo is successful
peerGetRoomKeysCallback	Callback for peerGetRoomKeys.
peerGlobalKeyChangedCallback	Called when a new value becomes available for a global watch key.
peerJoinRoomCallback	Callback for the following functions: peerJoinTitleRoom, peerJoinGroupRoom, peerJoinStagingRoom, peerJoinStagingRoomByIP, peerCreateStagingRoom[WithSocket].
peerKickedCallback	Callback when a local player was kicked from a room.
peerListGroupRoomsCallback	Callback for peerListGroupRooms
peerListingGamesCallback	Callback for peerStartListingGames
peerNewPlayerListCallback	Callback when the entire player list for the specified room has been updated
peerNickErrorCallback	

	Callback for peerConnect.
peerPingCallback	Callback when an updated ping for a player was just received.
peerPlayerChangedNickCallback	Callback when a player in one of the rooms changes his/her nick.
peerPlayerFlagsChangedCallback	Callback when a player's flags have changed in the room
peerPlayerInfoCallback	Callback when the IP and ProfileID for this player has just been received.
peerPlayerJoinedCallback	Callback when a player joins a room
peerPlayerLeftCallback	Callback when a player leaves a room.
peerPlayerMessageCallback	Callback called when a private message is received from another player
peerPlayerUTMCallback	Called when a private UTM is received from another player.
peerQRAddErrorCallback	Callback when reporting a game, this callback is called if there was an error with server reporting.
peerQRCountCallback	

	Callback when reporting a game, this callback is used to get a count of the number of players or teams.
peerQRKeyListCallback	Called when reporting a game, this callback is used to get a list of keys the application will report.
peerQRNatNegotiateCallback	Called when a nat-negotiate cookie is received.
peerQRPlayerKeyCallback	Called when getting values for any player keys the game is reporting.
peerQRPublicAddressCallback	Called when hosting a server with the server's public reporting address.
peerQRServerKeyCallback	Called when a server key is requested during a hosted game.
peerQRTeamKeyCallback	Callback is used to get values for any team keys the game is reporting.
peerReadyChangedCallback	Called when a player's ready state changes. All players default to not ready.
peerRoomKeyChangedCallback	Called when a new value becomes available for a room watch key, or when a broadcast key changes.
peerRoomMessageCallback	

Called when a message is sent to a room the local player is in.

[peerRoomModeChangedCallback](#)

Called when a room's mode changes.

[peerRoomNameChangedCallback](#)

Called when a room name changes

[peerRoomUTMCallback](#)

Called when a UTM is sent to a room the local player is in.

peerAuthenticateCDKeyCallback

Called when peerAuthenticateCDKey and attempt to authenticate the CD-Key is finished.

```
typedef void (*peerAuthenticateCDKeyCallback)(  
    PEER peer,  
    int result,  
    const gsi_char * message,  
    void * param );
```

Routine	Required Header	Distribution
peerAuthenticateCDKeyCallback	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object.

result

[in] Indicates the result of the attempt

message

[in] A text message representing the result

param

[in] Pointer to user data. This is optional and will be passed unmodified to the callback function.

Remarks

The **peerAuthenticateCDKeyCallback** function gets called when an attempt to authenticate a CD key is finished. If the result has a value of 1, the CD key was authenticated. Otherwise, the CD key was not authenticated.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_
peerAuthenticateCDKeyCallback	peerAuthenticateCDKeyCallbackA	peer.

peerAuthenticateCDKeyCallbackW and **peerAuthenticateCDKeyCallbackA** are UNICODE and ANSI mapped versions of **peerAuthenticateCDKeyCallback**. The arguments of **peerAuthenticateCDKeyCallbackA** are ANSI strings; those of **peerAuthenticateCDKeyCallbackW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

peerAutoMatchRateCallback

Called when rating the server for determining the best match.

```
typedef int (*peerAutoMatchRateCallback)(  
    PEER peer,  
    SBServer match,  
    void * param );
```

Routine	Required Header	Distribution
peerAutoMatchRateCallback	<peer.h>	SDKZIP

Return Value

Parameters

peer

[in] Initialized peer object

match

[in] A possible match for a Server

param

[in] Pointer to user data. This is optional and will be passed unmodified to the callback function.

Remarks

The **peerAutoMatchRateCallback** function is called one or more times for each game found. This allows the application to assign a rating to each game, and is used to determine the best fit for the user. The rating value should be calculated based on correlation between user preferred settings and actual game settings.

Section Reference: [Gamespy Peer SDK](#)

peerAutoMatchStatusCallback

Called when the Automatch state has changed.

```
typedef void (*peerAutoMatchStatusCallback)(  
    PEER peer,  
    PEERAutoMatchStatus status,  
    void *param );
```

Routine	Required Header	Distribution
peerAutoMatchStatusCallback	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object

status

[in] The current status

param

[in] Pointer to user data. This is optional and will be passed unmodified to the callback function.

Remarks

The **peerAutoMatchStatusCallback** function is called when the AutoMatch state changes. For example, when a game is joined or the user begins hosting a game. Refer to the status descriptions below for more information.

Section Reference: [Gamespy Peer SDK](#)

peerChangeNickCallback

Callback called after peerChangeNick attempt is finished.

```
typedef void (*peerChangeNickCallback)(  
    PEER peer,  
    PEERBool success,  
    const gsi_char * oldNick,  
    const gsi_char * newNick,  
    void * param );
```

Routine	Required Header	Distribution
peerChangeNickCallback	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object

success

[in] PEERTrue if successful, PEERFalse if failure

oldNick

[in] The nickname to be corrected or verified

newNick

[in] Corrected nickname. May be the same as oldNick if no issues are detected

param

[in] Pointer to user data. This is optional and will be passed unmodified to the callback function.

Remarks

If success is true, the attempt succeeded, and "newNick" is the local user's new nickname. If success is false, the attempt failed, and the local user's nick is still "oldNick".

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Def
peerChangeNickCallback	peerChangeNickCallbackA	peerChangeNickCa

peerChangeNickCallbackW and **peerChangeNickCallbackA** are UNICODE and ANSI mapped versions of **peerChangeNickCallback**. The arguments of **peerChangeNickCallbackA** are ANSI strings; those of **peerChangeNickCallbackW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

peerConnectCallback

Callback for peerConnect.

```
typedef void (*peerConnectCallback)(  
    PEER peer,  
    PEERBool success,  
    int failureReason,  
    void * param );
```

Routine	Required Header	Distribution
peerConnectCallback	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object.

success

[in] PEERTrue if connection was successful, PEERFalse if connection failed

failureReason

[in] int value giving reason for failure

param

[in] Pointer to user data. This is optional and will be passed unmodified to the callback function.

Remarks

The **peerConnectCallback** function will notify the application of a successful connection or a failure. Based on the "success" parameter the "failureReason" will tell the application why a connection failed. The values for the "failureReason" are:

PEER_DISCONNECTED

Unable to connect to the server, or disconnected during the attempt.

PEER_NICK_ERROR

There was a nick error that was not handled.

PEER_LOGIN_FAILED

The login info passed to peerConnectLogin was invalid.

Section Reference: [Gamespy Peer SDK](#)

peerCrossPingCallback

Callback for updated cross-ping between two players in the staging room.

```
typedef void (*peerCrossPingCallback)(  
    PEER peer,  
    const gsi_char * nick1,  
    const gsi_char * nick2,  
    int crossPing,  
    void * param );
```

Routine	Required Header	Distribution
peerCrossPingCallback	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object

nick1

[in] The first player's nick

nick2

[in] The second player's nick

crossPing

[in] The cross ping between the two players

param

[in] Pointer to user data. This is optional and will be passed unmodified to the callback function.

Remarks

This is the most recent ping from the first player (nick1) to the second player (nick2), as reported by the first player. This ordering information is not retained (i.e., peer stores the pings between sets of players, not each player's ping to each other player). To get the average ping between two players, use `peerGetPlayersCrossPing()`.

Section Reference: [Gamespy Peer SDK](#)

peerDisconnectedCallback

Called when a local player has been disconnected from the chat server for any reason.

```
typedef void (*peerDisconnectedCallback)(  
    PEER peer,  
    const gsi_char *reason,  
    void *param );
```

Routine	Required Header	Distribution
peerDisconnectedCallback	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object

reason

[in] The string containing the reason for disconnection

param

[in] Pointer to user data. This is optional and will be passed unmodified to the callback function.

Remarks

The **peerDisconnectedCallback** is called when the connection to the server gets disconnected, either from a call to `peerDisconnect()`, a lost connection, or getting killed by the server. To connect again, just use `peerConnect()`. After reconnecting, any rooms the user was in will need to be rejoined.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE I
peerDisconnectedCallback	peerDisconnectedCallbackA	peerDisconnecte

peerDisconnectedCallbackW and **peerDisconnectedCallbackA** are UNICODE and ANSI mapped versions of **peerDisconnectedCallback**. The arguments of **peerDisconnectedCallbackA** are ANSI strings; those of **peerDisconnectedCallbackW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

peerEnumPlayersCallback

Called for peerEnumPlayers for each player.

```
typedef void (*peerEnumPlayersCallback)(  
    PEER peer,  
    PEERBool success,  
    RoomType roomType,  
    int index,  
    const gsi_char *nick,  
    int flags,  
    void *param );
```

Routine	Required Header	Distribution
peerEnumPlayersCallback	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object

success

[in] PEERTrue if successful, or PEERFalse if failure

roomType

[in] Can be either TitleRoom, GroupRoom or StagingRoom

index

[in] The index of the current player being enumerated

nick

[in] The Chat nickname of that player

flags

[in] The Flags of that player

param

[in] Pointer to user data. This is optional and will be passed unmodified to the callback function.

Remarks

Called for each player in a room being enumerated, and once when finished, with "index" set to -1 and "nick" set to NULL. The index is not an identifier of any sort, its just a way to count the number of players that have been enumerated. It is not persistant in any way.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE D
peerEnumPlayersCallback	peerEnumPlayersCallbackA	peerEnumPlayers

peerEnumPlayersCallbackW and **peerEnumPlayersCallbackA** are UNICODE and ANSI mapped versions of **peerEnumPlayersCallback**. The arguments of **peerEnumPlayersCallbackA** are ANSI strings; those of **peerEnumPlayersCallbackW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

peerGameStartedCallback

Called when the host of a staging room launches the game.

```
typedef void (*peerGameStartedCallback)(  
    PEER peer,  
    SBServer server,  
    const gsi_char *message,  
    void *param );
```

Routine	Required Header	Distribution
peerGameStartedCallback	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object

server

[in] A valid SBServer object

message

[in] The message sent by the host

param

[in] Pointer to user data. This is optional and will be passed unmodified to the callback function.

Remarks

This is a notice to the other players in the room that the game is starting.
The host does not receive this callback.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE D
peerGameStartedCallback	peerGameStartedCallbackA	peerGameStarted

peerGameStartedCallbackW and **peerGameStartedCallbackA** are UNICODE and ANSI mapped versions of **peerGameStartedCallback**. The arguments of **peerGameStartedCallbackA** are ANSI strings; those of **peerGameStartedCallbackW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

See Also: [peerStartGame](#)

peerGetGlobalKeysCallback

Callback for peerGetPlayerGlobalKeys() and peerGetRoomGlobalKeys().

```
typedef void (*peerGetGlobalKeysCallback)(  
    PEER peer,  
    PEERBool success,  
    const gsi_char *nick,  
    int num,  
    const gsi_char **keys,  
    const gsi_char **values,  
    void *param );
```

Routine	Required Header	Distribution
peerGetGlobalKeysCallback	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object

success

[in] PEERTrue if successful, PEERFalse if failure

nick

[in] The player's nickname

num

[in] Number of key/value pairs in the array

keys

[in] Array of key names whose values were retrieved

values

[in] Array of values retrieved

param

[in] Pointer to user data. This is optional and will be passed unmodified to the callback function.

Remarks

Called with a player's global keys in response to either `peerGetPlayerGlobalKeys()` or `peerGetRoomGlobalKeys()`.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE
<code>peerGetGlobalKeysCallback</code>	<code>peerGetGlobalKeysCallbackA</code>	<code>peerGetGlobalKeysCallbackW</code>

`peerGetGlobalKeysCallbackW` and `peerGetGlobalKeysCallbackA` are UNICODE and ANSI mapped versions of `peerGetGlobalKeysCallback`. The arguments of `peerGetGlobalKeysCallbackA` are ANSI strings; those of `peerGetGlobalKeysCallbackW` are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

See Also: [peerGetPlayerGlobalKeys](#), [peerGetRoomGlobalKeys](#)

peerGetPlayerInfoCallback

Called after an attempt to peerGetPlayerInfo is successful.

```
typedef void (*peerGetPlayerInfoCallback)(  
    PEER peer,  
    PEERBool success,  
    const gsi_char *nick,  
    unsigned int IP,  
    int profileID,  
    void *param );
```

Routine	Required Header	Distribution
peerGetPlayerInfoCallback	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object

success

[in] PEERTrue if successful, PEERFalse if failure

nick

[in] The player's nickname that information is being requested for

IP

[in] IP address in string form: "xxx.xxx.xxx.xxx"

profileID

[in] The player's profile ID

param

[in] Pointer to user data. This is optional and will be passed unmodified to the callback function.

Remarks

The **peerGetPlayerInfoCallback** function will have the information requested by `peerGetPlayerInfo`. Any success will be determined by the "success" parameter. If the player did not exist, if the nick is invalid, or peer is not connected, the function will have a failure denoted by "success".

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE I
peerGetPlayerInfoCallback	peerGetPlayerInfoCallbackA	peerGetPlayerIn

peerGetPlayerInfoCallbackW and **peerGetPlayerInfoCallbackA** are UNICODE and ANSI mapped versions of **peerGetPlayerInfoCallback**. The arguments of **peerGetPlayerInfoCallbackA** are ANSI strings; those of **peerGetPlayerInfoCallbackW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

peerGetRoomKeysCallback

Callback for peerGetRoomKeys.

```
typedef void (*peerGetRoomKeysCallback)(  
    PEER peer,  
    PEERBool success,  
    RoomType roomType,  
    const gsi_char *nick,  
    int num,  
    gsi_char ** keys,  
    gsi_char ** values,  
    void * param );
```

Routine	Required Header	Distribution
peerGetRoomKeysCallback	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object

success

[in] PEERTrue if successful, PEERFalse if failure

roomType

[in] Can be either TitleRoom, GroupRoom or StagingRoom

nick

[in] The player's nickname

num

[in] The Number of key/value pairs in the array

keys

[in] Array of key names whose values will be retrieved

values

[in] Array of values to set

param

[in] Pointer to user data. This is optional and will be passed unmodified to the callback function.

Remarks

The **peerGetRoomKeysCallback** function is called when peer wants to obtain the room keys for a specified room. If nick is NULL, these are keys for the room. Otherwise, they are keys for a player in the room.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE
peerGetRoomKeysCallback	peerGetRoomKeysCallbackA	peerGetRoomKeysCallbackW

peerGetRoomKeysCallbackW and **peerGetRoomKeysCallbackA** are UNICODE and ANSI mapped versions of **peerGetRoomKeysCallback**. The arguments of **peerGetRoomKeysCallbackA** are ANSI strings; those of **peerGetRoomKeysCallbackW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

See Also: [peerGetRoomKeys](#)

peerGlobalKeyChangedCallback

Called when a new value becomes available for a global watch key.

```
typedef void (*peerGlobalKeyChangedCallback)(  
    PEER peer,  
    const gsi_char *nick,  
    const gsi_char *key,  
    const gsi_char *value,  
    void *param );
```

Routine	Required Header	Distribution
peerGlobalKeyChangedCallback	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object

nick

[in] The player's nickname

key

[in] The name of this key

value

[in] The value of this key

param

[in] Pointer to user data. This is optional and will be passed unmodified to the callback function.

Remarks

The **peerGlobalKeyChangedCallback** is called for watch keys when a room is joined, for watch keys when another player joins, and for any newly set watch keys.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_
peerGlobalKeyChangedCallback	peerGlobalKeyChangedCallbackA	peerG

peerGlobalKeyChangedCallbackW and **peerGlobalKeyChangedCallbackA** are UNICODE and ANSI mapped versions of **peerGlobalKeyChangedCallback**. The arguments of **peerGlobalKeyChangedCallbackA** are ANSI strings; those of **peerGlobalKeyChangedCallbackW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

peerJoinRoomCallback

Callback for the following functions: peerJoinTitleRoom, peerJoinGroupRoom, peerJoinStagingRoom, peerJoinStagingRoomByIP, peerCreateStagingRoom[WithSocket].

```
typedef void (*peerJoinRoomCallback)(  
    PEER peer,  
    PEERBool success,  
    PEERJoinResult result,  
    RoomType roomType,  
    void *param );
```

Routine	Required Header	Distribution
peerJoinRoomCallback	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object

success

[in] PEERTrue if successful, PEERFalse if failure

result

[in] Indicates the result of the attempt

roomType

[in] Can be either TitleRoom, GroupRoom or StagingRoom

param

[in] Pointer to user data. This is optional and will be passed unmodified to the callback function.

Remarks

The **peerJoinRoomCallback** is called when an attempt to join or create a room has finished. If successful, the local player is now in that room, and will be until he either leaves (with `peerLeaveRoom()`), is kicked, or the connection is disconnected. If success is `PEERFalse`, use result to check the reason for the failure. If result is `PEERBadPassword` the user can be prompted to enter a password, and then the join can be attempted again.

Section Reference: [Gamespy Peer SDK](#)

peerKickedCallback

Callback when a local player was kicked from a room.

```
typedef void (*peerKickedCallback)(  
    PEER peer,  
    RoomType roomType,  
    const gsi_char *nick,  
    const gsi_char *reason,  
    void *param );
```

Routine	Required Header	Distribution
peerKickedCallback	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object

roomType

[in] Can be either TitleRoom, GroupRoom or StagingRoom.

nick

[in] the player's nickname

reason

[in] An optional explanation string that gives a reason for being kicked

param

[in] Pointer to user data. This is optional and will be passed unmodified to the callback function.

Remarks

The **peerKickedCallback** is used to notify the local player that s/he was kicked. As the player has already been removed from the room, there is no need to call `peerLeaveRoom()`.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
peerKickedCallback	peerKickedCallbackA	peerKickedCallbackW

peerKickedCallbackW and **peerKickedCallbackA** are UNICODE and ANSI mapped versions of **peerKickedCallback**. The arguments of **peerKickedCallbackA** are ANSI strings; those of **peerKickedCallbackW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

peerListGroupRoomsCallback

Callback for peerListGroupRooms.

```
typedef void (*peerListGroupRoomsCallback)(  
    PEER peer,  
    PEERBool success,  
    int groupID,  
    SBServer server,  
    const gsi_char * name,  
    int numWaiting,  
    int maxWaiting,  
    int numGames,  
    int numPlaying,  
    void * param );
```

Routine	Required Header	Distribution
peerListGroupRoomsCallback	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object

success

[in] PEERTrue if successful, PEERFalse if failure

groupID

[in] The group ID of the current group room

server

[in] A valid SBServer object associated with this group room

name

[in] The name of the group room

numWaiting

[in] The number number of players in the room

maxWaiting

[in] The maximum number of players allowed in the room

numGames

[in] The number of games in the room

numPlaying

[in] The number of players already in a game for the room

param

[in] Pointer to user data. This is optional and will be passed unmodified to the callback function.

Remarks

The **peerListGroupRoomsCallback** function gets called once for each group room when listing group rooms. After this has been called for each group room, it will be called one more time with "groupID" set to 0.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNIC
peerListGroupRoomsCallback	peerListGroupRoomsCallbackA	peerListGr

peerListGroupRoomsCallbackW and **peerListGroupRoomsCallbackA** are UNICODE and ANSI mapped versions of **peerListGroupRoomsCallback**. The arguments of **peerListGroupRoomsCallbackA** are ANSI strings; those of **peerListGroupRoomsCallbackW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

peerListingGamesCallback

Callback for peerStartListingGames.

```
typedef void (*peerListingGamesCallback)(  
    PEER peer,  
    PEERBool success,  
    const gsi_char *name,  
    SBServer server,  
    PEERBool staging,  
    int msg,  
    int progress,  
    void *param );
```

Routine	Required Header	Distribution
peerListingGamesCallback	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object

success

[in] PEERTrue if successful, PEERFalse if failure

name

[in] The name of the game

server

[in] The valid SBServer object a game is associated with

staging

[in] PEERTrue if staging,

msg

[in] A message code

progress

[in] A progress of the initial listing

param

[in] Pointer to user data. This is optional and will be passed unmodified to the callback function.

Remarks

The **peerListingGamesCallback** function is called with info on games being listed. It is used to maintain a list of running games and staging rooms. The server object is a unique way of identifying each game. It can also be used with the calls in the "SBServer Object Functions" section of `sb_serverbrowsing.h` to find out more info about the server.

If "staging" is true, the game hasn't started yet, it's still in the staging room. Use `peerJoinStagingRoom()` to join the staging room. Or, if staging is false, use the server object to get the game's IP and port to join with.

The "password" key will be set to 1 for games that are passworded. This can be checked with `ServerGetIntValue(server, "password", 0)`.

The type of message this is.

PEER_CLEAR:

Clear the list. This has the same effect as if this was called with **PEER_REMOVE** for every server listed.

PEER_ADD:

This is a new server. Add it to the list.

PEER_UPDATE:

This server is already on the list, and its been updated.

PEER_REMOVE:

Remove this server from the list. The server object for this server should not be used again after this callback returns.

PEER_COMPLETE:

The initial listing of servers has completed, and dynamic changes will now be received.

When first starting to list games, an initial list of current games is received, then updated as new game are started and old games are updated or removed. While the initial listing is happening, this lets the program know what percentage of the initial list has been added so far. It will start at 0 with the **PEER_CLEAR** message, then rise up to 100 with the **PEER_COMPLETE** message. When it reaches 100, it will stay there until the listing is stopped.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE I
peerListingGamesCallback	peerListingGamesCallbackA	peerListingGame

peerListingGamesCallbackW and **peerListingGamesCallbackA** are UNICODE and ANSI mapped versions of **peerListingGamesCallback**. The arguments of **peerListingGamesCallbackA** are ANSI strings; those of **peerListingGamesCallbackW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

peerNewPlayerListCallback

Callback when the entire player list for the specified room has been updated.

```
typedef void (*peerNewPlayerListCallback)(  
    PEER peer,  
    RoomType roomType,  
    void * param );
```

Routine	Required Header	Distribution
peerNewPlayerListCallback	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object

roomType

[in] Can be either TitleRoom, GroupRoom or StagingRoom.

param

[in] Pointer to user data. This is optional and will be passed unmodified to the callback function.

Remarks

The **peerNewPlayerListCallback** gets generated after turning off quiet mode. It serves as notice that the player list for this room has been updated. To get the new list, use `peerEnumPlayers()`.

Section Reference: [Gamespy Peer SDK](#)

peerNickErrorCallback

Callback for peerConnect.

```
typedef void (*peerNickErrorCallback)(  
    PEER peer,  
    int type,  
    const gsi_char *nick,  
    int numSuggestedNicks,  
    const gsi_char **suggestedNicks,  
    void *param );
```

Routine	Required Header	Distribution
peerNickErrorCallback	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object

type

[in] One of the predefined error types

nick

[in] The player's nickname passed to peerConnect or peerRegisterUniqueNick

numSuggestedNicks

[in] The number of suggested nicknames

suggestedNicks

[in] A List of suggested nicknames

param

[in] Pointer to user data. This is optional and will be passed unmodified to the callback function.

Remarks

If this function is called with type equal to PEER_IN_USE or PEER_INVALID, there was an error with the nick that was passed to peerConnect() or peerRetryWithNick(). The connection attempt is put on hold until peerRetryWithNick() is called. It does not need to be called immediately, but should be called within a reasonable amount of time, or the connection attempt may time out. If the nick passed to peerRetryWithNick fails, this callback will be called again (and as many times as needed) until the server accepts a nick. To stop attempting reconnects, call peerRetryWithNick() with a NULL or empty nickname. That will cause the connectCallback passed to peerConnect() to be called with success set to false.

If this function is called with type equal to PEER_UNIQUENICK_EXPIRED or PEER_NO_UNIQUENICK, then there was a problem with the uniquenick associated with the profile passed to peerConnectLogin. The connection attempt is put on hold until peerRegisterUniqueNick() is called. It does not need to be called immediately - the connection will stay alive until it is called. If the uniquenick passed to peerRegisterUniqueNick is invalid or already being used, then this callback will be called again with a type of PEER_INVALID_UNIQUENICK, and with suggestedNicks member filled with an array of suggested uniquenicks (based on the uniquenick passed to peerRegisterUniqueNick). In that case, peerRegisterUniqueNick should be called again with a new uniquenick.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
peerNickErrorCallback	peerNickErrorCallbackA	peerNickErrorCallbackW

peerNickErrorCallbackW and **peerNickErrorCallbackA** are UNICODE and ANSI mapped versions of **peerNickErrorCallback**. The arguments of **peerNickErrorCallbackA** are ANSI strings; those of **peerNickErrorCallbackW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

peerPingCallback

Callback when an updated ping for a player was just received.

```
typedef void (*peerPingCallback)(  
    PEER peer,  
    const gsi_char *nick,  
    int ping,  
    void *param );
```

Routine	Required Header	Distribution
peerPingCallback	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object

nick

[in] The player's nickname

ping

[in] The Ping value

param

[in] Pointer to user data. This is optional and will be passed unmodified to the callback function.

Remarks

The **peerPingCallback** will have the latest ping for a player. This is the value of the most recent ping of this player. To get the average ping for this player, use `peerGetPlayerPing`.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
peerPingCallback	peerPingCallbackA	peerPingCallbackW

peerPingCallbackW and **peerPingCallbackA** are UNICODE and ANSI mapped versions of **peerPingCallback**. The arguments of **peerPingCallbackA** are ANSI strings; those of **peerPingCallbackW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

peerPlayerChangedNickCallback

Callback when a player in one of the rooms changes his/her nick.

```
typedef void (*peerPlayerChangedNickCallback)(  
    PEER peer,  
    RoomType roomType,  
    const gsi_char *oldNick,  
    const gsi_char *newNick,  
    void *param );
```

Routine	Required Header	Distribution
peerPlayerChangedNickCallback	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object

roomType

[in] Can be either TitleRoom, GroupRoom or StagingRoom.

oldNick

[in] The nickname to be corrected or verified

newNick

[in] Corrected nickname. May be the same as oldNick if no issues are detected

param

[in] Pointer to user data. This is optional and will be passed unmodified to the callback function.

Remarks

The **peerPlayerChangedNickCallback** is called for any changes a player makes to his/her nick in a specified room. If in multiple rooms with the same player, this callback will be called for each common room when that player changes his nick.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_
peerPlayerChangedNickCallback	peerPlayerChangedNickCallbackA	peer

peerPlayerChangedNickCallbackW and **peerPlayerChangedNickCallbackA** are UNICODE and ANSI mapped versions of **peerPlayerChangedNickCallback**. The arguments of **peerPlayerChangedNickCallbackA** are ANSI strings; those of **peerPlayerChangedNickCallbackW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

peerPlayerFlagsChangedCallback

Callback when a player's flags have changed in the room.

```
typedef void (*peerPlayerFlagsChangedCallback)(  
    PEER peer,  
    RoomType roomType,  
    const gsi_char *nick,  
    int oldFlags,  
    int newFlags,  
    void *param );
```

Routine	Required Header	Distribution
peerPlayerFlagsChangedCallback	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object

roomType

[in] Can be either TitleRoom, GroupRoom or StagingRoom.

nick

[in] The player's nickname

oldFlags

[in] Old flags value

newFlags

[in] New flags value

param

[in] Pointer to user data. This is optional and will be passed unmodified to the callback function.

Remarks

The **peerPlayerFlagsChangedCallback** is called when a player's flags change in the specified room. The flags each represent one bit in the "flags" integer. The new flags can also trigger a change in the chat, where a player can become an operator, leave a room, enter a room, stage in a room.

The flags are:

PEER_FLAG_STAGING: the player is in a staging room.

PEER_FLAG_READY: the player is readied up for a game.

PEER_FLAG_PLAYING: the player is playing a game.

PEER_FLAG_AWAY: the player is away.

PEER_FLAG_HOST: the player is the host of the room.

PEER_FLAG_OP: the player is an op (+o) in this room.

PEER_FLAG_VOICE: the player has voice (+v) in this room.

This function will fail if no title is set.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	G
peerPlayerFlagsChangedCallback	peerPlayerFlagsChangedCallbackA	pe

peerPlayerFlagsChangedCallbackW and **peerPlayerFlagsChangedCallbackA** are UNICODE and ANSI mapped versions of **peerPlayerFlagsChangedCallback**. The arguments of **peerPlayerFlagsChangedCallbackA** are ANSI strings; those of **peerPlayerFlagsChangedCallbackW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

peerPlayerInfoCallback

Callback when the IP and ProfileID for this player has just been received.

```
typedef void (*peerPlayerInfoCallback)(  
    PEER peer,  
    RoomType roomType,  
    const gsi_char *nick,  
    unsigned int IP,  
    int profileID,  
    void *param );
```

Routine	Required Header	Distribution
peerPlayerInfoCallback	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object

roomType

[in] Can be either TitleRoom, GroupRoom or StagingRoom.

nick

[in] The player's chat nickname

IP

[in] The player's IP address in string form: "xxx.xxx.xxx.xxx"

profileID

[in] The player's Profile ID

param

[in] Pointer to user data. This is optional and will be passed unmodified to the callback function.

Remarks

After joining a room, the **peerPlayerInfoCallback** function will be called for each player in the room who is using Peer with his IP and profile ID. Then it will be called one more time with nick set to NULL. This info is immediately available for anyone who joins the room after the local player.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
peerPlayerInfoCallback	peerPlayerInfoCallbackA	peerPlayerInfoCallbackW

peerPlayerInfoCallbackW and **peerPlayerInfoCallbackA** are UNICODE and ANSI mapped versions of **peerPlayerInfoCallback**. The arguments of **peerPlayerInfoCallbackA** are ANSI strings; those of **peerPlayerInfoCallbackW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

peerPlayerJoinedCallback

Callback when a player joins a room.

```
typedef void (*peerPlayerJoinedCallback)(  
    PEER peer,  
    RoomType roomType,  
    const gsi_char *nick,  
    void *param );
```

Routine	Required Header	Distribution
peerPlayerJoinedCallback	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object

roomType

[in] Can be either TitleRoom, GroupRoom or StagingRoom.

nick

[in] The player's chat nickname

param

[in] Pointer to user data. This is optional and will be passed unmodified to the callback function.

Remarks

The **peerPlayerJoinedCallback** function gets called when a player joins the specified room. This function can be used as a place for report player joins.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE D
peerPlayerJoinedCallback	peerPlayerJoinedCallbackA	peerPlayerJoined

peerPlayerJoinedCallbackW and **peerPlayerJoinedCallbackA** are UNICODE and ANSI mapped versions of **peerPlayerJoinedCallback**. The arguments of **peerPlayerJoinedCallbackA** are ANSI strings; those of **peerPlayerJoinedCallbackW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

peerPlayerLeftCallback

Callback when a player leaves a room.

```
typedef void (*peerPlayerLeftCallback)(  
    PEER peer,  
    RoomType roomType,  
    const gsi_char *nick,  
    const gsi_char *reason,  
    void *param );
```

Routine	Required Header	Distribution
peerPlayerLeftCallback	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object

roomType

[in] Can be either TitleRoom, GroupRoom or StagingRoom.

nick

[in] The player's chat nickname

reason

[in] An optional explanation string

param

[in] Pointer to user data. This is optional and will be passed unmodified to the callback function.

Remarks

The **peerPlayerLeftCallback** function is called when a player decides to leave the specified room. There is an optional "reason" that could be given.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
peerPlayerLeftCallback	peerPlayerLeftCallbackA	peerPlayerLeftCallbackW

peerPlayerLeftCallbackW and **peerPlayerLeftCallbackA** are UNICODE and ANSI mapped versions of **peerPlayerLeftCallback**. The arguments of **peerPlayerLeftCallbackA** are ANSI strings; those of **peerPlayerLeftCallbackW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

peerPlayerMessageCallback

Callback called when a private message is received from another player.

```
typedef void (*peerPlayerMessageCallback)(  
    PEER peer,  
    const gsi_char *nick,  
    const gsi_char *message,  
    MessageType messageType,  
    void *param );
```

Routine	Required Header	Distribution
peerPlayerMessageCallback	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object.

nick

[in] The player's chat nickname

message

[in] The message sent by the player.

messageType

[in] The type of message sent. Most commonly NormalMessage or ActionMessage.

param

[in] Pointer to user data. This is optional and will be passed unmodified to the callback function.

Remarks

The **peerPlayerMessageCallback** function is called when a remote player sends the local player a message. The message can be of two types, namely NormalMessage or ActionMessage.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE
<code>peerPlayerMessageCallback</code>	<code>peerPlayerMessageCallbackA</code>	<code>peerPlayerM</code>

`peerPlayerMessageCallbackW` and `peerPlayerMessageCallbackA` are UNICODE and ANSI mapped versions of `peerPlayerMessageCallback`. The arguments of `peerPlayerMessageCallbackA` are ANSI strings; those of `peerPlayerMessageCallbackW` are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

peerPlayerUTMCallback

Called when a private UTM is received from another player.

```
typedef void (*peerPlayerUTMCallback)(  
    PEER peer,  
    const gsi_char *nick,  
    const gsi_char *command,  
    const gsi_char *parameters,  
    PEERBool authenticated,  
    void *param );
```

Routine	Required Header	Distribution
peerPlayerUTMCallback	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object.

nick

[in] The player's chat nickname.

command

[in] The raw UTM command sent for this message.

parameters

[in] The parameters sent along with the command.

authenticated

[in] True if authenticated the server.

param

[in] Pointer to user data. This is optional and will be passed unmodified to the callback function.

Remarks

The **peerPlayerUTMCallback** function is called after receiving a UTM from a remote player. The command is a short string identifying this UTM (e.g., RQI, NFO, GML). Ignore any unrecognized UTM, as internal Peer UTMs go through this callback, as do application UTMs.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defin
peerPlayerUTMCallback	peerPlayerUTMCallbackA	peerPlayerUTMCallba

peerPlayerUTMCallbackW and **peerPlayerUTMCallbackA** are UNICODE and ANSI mapped versions of **peerPlayerUTMCallback**. The arguments of **peerPlayerUTMCallbackA** are ANSI strings; those of **peerPlayerUTMCallbackW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

peerQRAddErrorCallback

Callback when reporting a game, this callback is called if there was an error with server reporting.

```
typedef void (*peerQRAddErrorCallback)(  
    PEER peer,  
    qr2_error_t error,  
    gsi_char *errorString,  
    void *param );
```

Routine	Required Header	Distribution
peerQRAddErrorCallback	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object

error

[in] A Qr2 error code when reporting fails

errorString

[in] The error in string form

param

[in] Pointer to user data. This is optional and will be passed unmodified to the callback function.

Remarks

The **peerQRAddErrorCallback** function is called when an error is flagged while reporting to the server. This callback is called while reporting if there is an error reporting the server. To see the possible error codes, look for the `qr2_error_t` enumeration in `qr2.h`.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Def
peerQRAddErrorCallback	peerQRAddErrorCallbackA	peerQRAddErrorCa

peerQRAddErrorCallbackW and **peerQRAddErrorCallbackA** are UNICODE and ANSI mapped versions of **peerQRAddErrorCallback**. The arguments of **peerQRAddErrorCallbackA** are ANSI strings; those of **peerQRAddErrorCallbackW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

peerQRCountCallback

Callback when reporting a game, this callback is used to get a count of the number of players or teams.

```
typedef int (*peerQRCountCallback)(  
    PEER peer,  
    qr2_key_type type,  
    void * param );
```

Routine	Required Header	Distribution
peerQRCountCallback	<peer.h>	SDKZIP

Return Value

Parameters

peer

[in] Initialized peer object

type

[in] A type of Qr2 keys. Can be either `key_server`, `key_player`, `key_team`.

param

[in] Pointer to user data. This is optional and will be passed unmodified to the callback function.

Remarks

The **peerQRCountCallback** function is called while hosting a game to get the number of players or teams. For example usage see the PeerTest sample. For a detailed explanation of how server reporting works, see the Query & Reporting 2 SDK documentation.

If in staging, and either not playing or peerStartGame was called with PEER_REPORT_PLAYERS set in the reporting options, then Peer will report the number of players, and this callback will not be called for a player count. Otherwise the application must report the number of players in this callback, and it must always report the number of teams (or 0 if not using teams).

Section Reference: [Gamespy Peer SDK](#)

peerQRKeyListCallback

Called when reporting a game, this callback is used to get a list of keys the application will report.

```
typedef void (*peerQRKeyListCallback)(  
    PEER peer,  
    qr2_key_type type,  
    qr2_keybuffer_t keyBuffer,  
    void * param );
```

Routine	Required Header	Distribution
peerQRKeyListCallback	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object

type

[in] A type of Qr2 keys. Can be either `key_server`, `key_player`, `key_team`.

keyBuffer

[in] The buffer to append the key to

param

[in] Pointer to user data. This is optional and will be passed unmodified to the callback function.

Remarks

The **peerQRKeyListCallback** function is called while hosting a game to list keys the application will report. For example usage see the PeerTest sample. For a detailed explanation of how server reporting works, see the Query & Reporting 2 SDK documentation.

Peer already registers certain keys, and these do not need to be registered by the application. Peer registers the following server keys: HOSTNAME_KEY, NUMPLAYERS_KEY, MAXPLAYERS_KEY, GAMEMODE_KEY, PASSWORD_KEY (if a password is set), and GROUPID_KEY (if in a group room). Peer also registers the player keys PLAYER__KEY and PING__KEY. Any other keys used by the application must be registered in this callback.

Section Reference: [Gamespy Peer SDK](#)

peerQRNatNegotiateCallback

Called when a nat-negotiate cookie is received.

```
typedef void (*peerQRNatNegotiateCallback)(  
    PEER peer,  
    int cookie,  
    void * param );
```

Routine	Required Header	Distribution
peerQRNatNegotiateCallback	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object

cookie

[in] Cookie received. Usually an integer value randomly generated by the sender.

param

[in] Pointer to user data. This is optional and will be passed unmodified to the callback function.

Remarks

The `peerQRNatNegotiationCallback` function is called when a nat-negotiate cookie is received. See the nat-negotiate documentation for more information.

Section Reference: [Gamespy Peer SDK](#)

peerQRPlayerKeyCallback

Called when getting values for any player keys the game is reporting.

```
typedef void (*peerQRPlayerKeyCallback)(  
    PEER peer,  
    int key,  
    int index,  
    qr2_buffer_t buffer,  
    void * param );
```

Routine	Required Header	Distribution
peerQRPlayerKeyCallback	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object

key

[in] The key for reporting information

index

[in] The array index of the player to report

buffer

[in] The buffer for data

param

[in] Pointer to user data. This is optional and will be passed unmodified to the callback function.

Remarks

The **peerQRPlayerKeyCallback** function is called while hosting a game to report values for player keys. For example usage see the PeerTest sample. For a detailed explanation of how server reporting works, see the Query & Reporting 2 SDK documentation.

If in staging, and either not playing or peerStartGame was called with PEER_REPORT_PLAYERS set in the reporting options, then Peer will report the PLAYER__KEY and PING__KEY keys, and the callback will not be called for these key. The application is responsible for reporting any other player keys and is also responsible for these keys when Peer does not report them. Any other keys the application reports must be registered with the peerQRKeyListCallback() (see below). The number of players is set with the peerQRCountCallback() see below).

Section Reference: [Gamespy Peer SDK](#)

peerQRPublicAddressCallback

Called when hosting a server with the server's public reporting address.

```
typedef void (*peerQRPublicAddressCallback)(  
    PEER peer,  
    unsigned int ip,  
    unsigned short port,  
    void * param );
```

Routine	Required Header	Distribution
peerQRPublicAddressCallback	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object

ip

[in] The IP address of the host in string form: "xxx.xxx.xxx.xxx"

port

[in] The Port number of the host

param

[in] Pointer to user data. This is optional and will be passed unmodified to the callback function.

Remarks

The **peerQRPublicAddressCallback** function is called when a host has been requested to send its public IP and public port to the requester.

Section Reference: [Gamespy Peer SDK](#)

peerQRServerKeyCallback

Called when a server key is requested during a hosted game.

```
typedef void (*peerQRServerKeyCallback)(  
    PEER peer,  
    int key,  
    qr2_buffer_t buffer,  
    void *param );
```

Routine	Required Header	Distribution
peerQRServerKeyCallback	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object

key

[in] The value associated with this key will be returned

buffer

[in] The Qr2 buffer for holding the data

param

[in] Pointer to user data. This is optional and will be passed unmodified to the callback function.

Remarks

The **peerQRServerKeyCallback** function is called while hosting a game to report values for server keys. For example usage see the PeerTest sample. For a detailed explanation of how server reporting works, see the Query & Reporting 2 SDK documentation.

If in staging, and either not playing or peerStartGame was called with PEER_REPORT_INFO set in the reporting options, then Peer will report the following keys: HOSTNAME_KEY, NUMPLAYERS_KEY, MAXPLAYERS_KEY, GAMEMODE_KEY (only if not playing), and PASSWORD_KEY, and the callback will not be called for these key. Peer will also always report the GROUPID_KEY if in a group room. The application is responsible for reporting any other server keys and is also responsible for these keys when Peer does not report them. Any other keys the application reports must be registered with the peerQRKeyListCallback() (see below).

Section Reference: [Gamespy Peer SDK](#)

peerQRTeamKeyCallback

Callback is used to get values for any team keys the game is reporting.

```
typedef void (*peerQRTeamKeyCallback)(  
    PEER peer,  
    int key,  
    int index,  
    qr2_buffer_t buffer,  
    void * param );
```

Routine	Required Header	Distribution
peerQRTeamKeyCallback	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object

key

[in] The key that will be returned

index

[in] The array index of the team requested

buffer

[in] The Qr2 buffer for data

param

[in] Pointer to user data. This is optional and will be passed unmodified to the callback function.

Remarks

The **peerQRTeamKeyCallback** is called while hosting a game to report values for team keys. For example usage see the PeerTest sample. For a detailed explanation of how server reporting works, see the Query & Reporting 2 SDK documentation.

Peer does not report any team keys. The application is responsible for reporting any team keys. Any keys the application reports must be registered with the `peerQRKeyListCallback()` (see below). The number of teams is set with the `peerQRCountCallback()` (see below).

Section Reference: [Gamespy Peer SDK](#)

peerReadyChangedCallback

Called when a player's ready state changes. All players default to not ready.

```
typedef void (*peerReadyChangedCallback)(  
    PEER peer,  
    const gsi_char *nick,  
    PEERBool ready,  
    void *param );
```

Routine	Required Header	Distribution
peerReadyChangedCallback	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object

nick

[in] The player's chat nickname

ready

[in] PEERTrue if ready, PEERFalse if not

param

[in] Pointer to user data. This is optional and will be passed unmodified to the callback function.

Remarks

The **peerReadyChangedCallback** is called whenever a player changes his/her ready status. This can only be done in a StagingRoom room type.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE
peerReadyChangedCallback	peerReadyChangedCallbackA	peerReadyC

peerReadyChangedCallbackW and **peerReadyChangedCallbackA** are UNICODE and ANSI mapped versions of **peerReadyChangedCallback**. The arguments of **peerReadyChangedCallbackA** are ANSI strings; those of **peerReadyChangedCallbackW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

peerRoomKeyChangedCallback

Called when a new value becomes available for a room watch key, or when a broadcast key changes.

```
typedef void (*peerRoomKeyChangedCallback)(  
    PEER peer,  
    RoomType roomType,  
    const gsi_char *nick,  
    const gsi_char *key,  
    const gsi_char *value,  
    void *param );
```

Routine	Required Header	Distribution
peerRoomKeyChangedCallback	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object

roomType

[in] Can be either TitleRoom, GroupRoom or StagingRoom.

nick

[in] The player's chat nickname whose keys changed

key

[in] The key that has changed

value

[in] The value that is associated with this key that has changed

param

[in] Pointer to user data. This is optional and will be passed unmodified to the callback function.

Remarks

The **peerRoomKeyChangedCallback** is called for watch keys when a room is joined, for watch keys when another player joins, for any newly set watch keys, and when a broadcast watch key is changed.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_U
peerRoomKeyChangedCallback	peerRoomKeyChangedCallbackA	peerR

peerRoomKeyChangedCallbackW and **peerRoomKeyChangedCallbackA** are UNICODE and ANSI mapped versions of **peerRoomKeyChangedCallback**. The arguments of **peerRoomKeyChangedCallbackA** are ANSI strings; those of **peerRoomKeyChangedCallbackW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

peerRoomMessageCallback

Called when a message is sent to a room the local player is in.

```
typedef void (*peerRoomMessageCallback)(  
    PEER peer,  
    RoomType roomType,  
    const gsi_char *nick,  
    const gsi_char *message,  
    MessageType messageType,  
    void *param );
```

Routine	Required Header	Distribution
peerRoomMessageCallback	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object

roomType

[in] Can be either TitleRoom, GroupRoom or StagingRoom.

nick

[in] The chat nickname of the player who sent the message

message

[in] The message that was sent from the player

messageType

[in] The type of message to send, most commonly NormalMessage or ActionMessage

param

[in] Pointer to user data. This is optional and will be passed unmodified to the callback function.

Remarks

The **peerRoomMessageCallback** function gets called when a player sends a message. This message is sent to everyone in the specified room. The message will be a plain text message.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICOD
peerRoomMessageCallback	peerRoomMessageCallbackA	peerRoomMe

peerRoomMessageCallbackW and **peerRoomMessageCallbackA** are UNICODE and ANSI mapped versions of **peerRoomMessageCallback**. The arguments of **peerRoomMessageCallbackA** are ANSI strings; those of **peerRoomMessageCallbackW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

peerRoomModeChangedCallback

Called when a room's mode changes.

```
typedef void (*peerRoomModeChangedCallback)(  
    PEER peer,  
    RoomType roomType,  
    CHATChannelMode * mode,  
    void * param );
```

Routine	Required Header	Distribution
peerRoomModeChangedCallback	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object

roomType

[in] Can be either TitleRoom, GroupRoom or StagingRoom.

mode

[in] The current mode for this room

param

[in] Pointer to user data. This is optional and will be passed unmodified to the callback function.

Remarks

The **peerRoomModeChangedCallback** is called as a host of the specified room changes a mode for the room. See chat.h in the Chat SDK for more information on channel modes.

Section Reference: [Gamespy Peer SDK](#)

peerRoomNameChangedCallback

Called when a room name changes.

```
typedef void (*peerRoomNameChangedCallback)(  
    PEER peer,  
    RoomType roomType,  
    void * param );
```

Routine	Required Header	Distribution
peerRoomNameChangedCallback	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object

roomType

[in] Can be either TitleRoom, GroupRoom or StagingRoom.

param

[in] Pointer to user data. This is optional and will be passed unmodified to the callback function.

Remarks

The **peerRoomNameChangedCallback** is called when the name of the specified room changes. This callback just serves as a notice. To get the actual name use `peerGetRoomName()`.

Section Reference: [Gamespy Peer SDK](#)

peerRoomUTMCallback

Called when a UTM is sent to a room the local player is in.

```
typedef void (*peerRoomUTMCallback)(  
    PEER peer,  
    RoomType roomType,  
    const gsi_char *nick,  
    const gsi_char *command,  
    const gsi_char *parameters,  
    PEERBool authenticated,  
    void *param );
```

Routine	Required Header	Distribution
peerRoomUTMCallback	<peer.h>	SDKZIP

Parameters

peer

[in] Initialized peer object

roomType

[in] Can be either TitleRoom, GroupRoom or StagingRoom.

nick

[in] The chat nickname of the player who sent the UTM

command

[in] The raw UTM command sent

parameters

[in] The parameters sent along with the UTM command

authenticated

[in] PEERTrue if authenticated, PEERFalse if otherwise

param

[in] Pointer to user data. This is optional and will be passed unmodified to the callback function.

Remarks

The **peerRoomUTMCallback** is called when a player in the specified room sends a UTM message. The command is a short string identifying this UTM (e.g., RQI, NFO, GML). Ignore any unrecognized UTM, as internal Peer UTMs go through this callback, as do application UTMs.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
peerRoomUTMCallback	peerRoomUTMCallbackA	peerRoomUTMCallbackW

peerRoomUTMCallbackW and **peerRoomUTMCallbackA** are UNICODE and ANSI mapped versions of **peerRoomUTMCallback**. The arguments of **peerRoomUTMCallbackA** are ANSI strings; those of **peerRoomUTMCallbackW** are wide-character strings.

Section Reference: [Gamespy Peer SDK](#)

Peer SDK Structures

[PEERCallbacks](#)

Structure that gets passed into peerInitialize()

PEERCallbacks

Structure that gets passed into peerInitialize().

```
typedef struct  
{  
    peerDisconnectedCallback disconnected;  
    peerRoomMessageCallback roomMessage;  
    peerRoomUTMCallback roomUTM;  
    peerRoomNameChangedCallback roomNameChanged;  
    peerRoomModeChangedCallback roomModeChanged;  
    peerPlayerMessageCallback playerMessage;  
    peerPlayerUTMCallback playerUTM;  
    peerReadyChangedCallback readyChanged;  
    peerGameStartedCallback gameStarted;  
    peerPlayerJoinedCallback playerJoined;  
    peerPlayerLeftCallback playerLeft;  
    peerKickedCallback kicked;  
    peerNewPlayerListCallback newPlayerList;  
    peerPlayerChangedNickCallback playerChangedNick;  
    peerPlayerInfoCallback playerInfo;  
    peerPlayerFlagsChangedCallback playerFlagsChanged;  
    peerPingCallback ping;  
    peerCrossPingCallback crossPing;  
    peerGlobalKeyChangedCallback globalKeyChanged;  
    peerRoomKeyChangedCallback roomKeyChanged;  
    peerQRServerKeyCallback qrServerKey;  
    peerQRPlayerKeyCallback qrPlayerKey;  
    peerQRTeamKeyCallback qrTeamKey;  
    peerQRKeyListCallback qrKeyList;  
    peerQRCountCallback qrCount;  
    peerQRAddErrorCallback qrAddError;  
    peerQRNatNegotiateCallback qrNatNegotiateCallback;  
    peerQRPublicAddressCallback qrPublicAddressCallback;  
    void * param;  
} PEERCallbacks;
```

Members

disconnected

Called when the chat connection has been disconnected by the server.

roomMessage

Called when a chat message has arrived in one of the rooms the developer is in.

roomUTM

Called when an under-the-table message has arrived in a room the developer is in.

roomNameChanged

Called when the name of a room the developer is in has changed.

roomModeChanged

Called when the mode changed in a room the developer is in.

playerMessage

Called when a private chat message from another player has been received.

playerUTM

Called when an under-the-table message has arrived from another player.

readyChanged

Called when another player in the same staging room as the user, has changed his ready status.

gameStarted

Called when the host in the staging room launches the game.

playerJoined

Called when a player has joined one of the rooms the local player has joined.

playerLeft

Called when a player has left one of the rooms the local player has joined.

kicked

Called when the local player has been kicked from a room.

newPlayerList

Called when the entire playerlist has been updated.

playerChangedNick

Called when player in one of the rooms changed his nick.

playerInfo

Called for all players (who are using peer) in a room shortly after joining.

playerFlagsChanged

for all players (who are using peer) in a room
Called when a player's flags have changed.

crossPing

An updated ping for a player, who may be in any room(s).

qrServerKey

Called for watch keys when a room is joined, for watch keys when another player joins, and for any newly set watch keys.

qrPlayerKey

Called to report QR player keys.

qrTeamKey

Called to report QR team keys.

qrKeyList

Called to get a list of keys to be reported.

qrCount

Called to get a count of the number of players or teams.

qrAddError

Called when there is an error reporting the server.

qrNatNegotiateCallback

Called when hosting a server and a nat-negotiate cookie is received.

qrPublicAddressCallback

Called when hosting a server with the server's public reporting address.

param

A pointer to data that will be passed into each of the callbacks when

triggered.

Section Reference: [Gamespy Peer SDK](#)

See Also: [peerInitialize](#)

Peer SDK Enumerations

MessageType	Types of messages.
PEERAutoMatchStatus	Possible status values passed to the peerAutoMatchStatusCallback. If PEERFailed, the match failed. Otherwise, this is the current status of the automatch attempt
PEERBool	Standard Boolean.
PEERJoinResult	Possible results when attempting to join a room. Passed into peerJoinRoomCallback().
RoomType	Types of rooms.

MessageType

Types of messages.

```
typedef enum  
{  
    NormalMessage,  
    ActionMessage,  
    NoticeMessage  
} MessageType;
```

Constants

NormalMessage

A normal chat message.

ActionMessage

An action message.

NoticeMessage

A notification message.

Section Reference: [Gamespy Peer SDK](#)

See Also: [peerMessagePlayer](#), [peerMessageRoom](#),
[peerPlayerMessageCallback](#), [peerRoomMessageCallback](#)

PEERAutoMatchStatus

Possible status values passed to the peerAutoMatchStatusCallback. If PEERFailed, the match failed. Otherwise, this is the current status of the automatch attempt.

```
typedef enum  
{  
    PEERFailed,  
    PEERSearching,  
    PEERWaiting,  
    PEERStaging,  
    PEERReady,  
    PEERComplete  
} PEERAutoMatchStatus;
```

Constants

PEERFailed

The automatch attempt failed.

PEERSearching

Searching for a match (active).

PEERWaiting

Waiting for a match (passive).

PEERStaging

In a staging room with at least one other player, possibly waiting for more.

PEERReady

All players are in the staging room, the game is ready to be launched.

PEERComplete

The game is launching, the automatch attempt is now complete. The player is still in the staging room.

Section Reference: [Gamespy Peer SDK](#)

PEERBool

Standard Boolean.

```
typedef enum  
{  
    PEERFalse,  
    PEERTrue  
} PEERBool;
```

Constants

PEERFalse
False.

PEERTrue
True.

Section Reference: [Gamespy Peer SDK](#)

PEERJoinResult

Possible results when attempting to join a room. Passed into `peerJoinRoomCallback()`.

typedef enum

{

PEERJoinSuccess,
PEERFullRoom,
PEERInviteOnlyRoom,
PEERBannedFromRoom,
PEERBadPassword,

PEERAlreadyInRoom,
PEERNoTitleSet,
PEERNoConnection,
PEERJoinFailed,

} PEERJoinResult;

Constants

PEERJoinSuccess

The room was joined.

PEERFullRoom

The room is full.

PEERInviteOnlyRoom

The room is invite only.

PEERBannedFromRoom

The local user is banned from the room.

PEERBadPassword

An incorrect password (or none) was given for a passworded room.

PEERAlreadyInRoom

The local user is already in or entering a room of the same type.

PEERNoTitleSet

Can't join a room if no title is set.

PEERNoConnection

Can't join a room if there's no chat connection.

PEERJoinFailed

Generic failure.

Section Reference: [Gamespy Peer SDK](#)

RoomType

Types of rooms.

```
typedef enum  
{  
    TitleRoom,  
    GroupRoom,  
    StagingRoom,  
    NumRooms  
} RoomType;
```

Constants

TitleRoom

The main room for a game.

GroupRoom

A room which is, in general, for a particular type of gameplay (team, DM, etc.).

StagingRoom

A room where players meet before starting a game.

NumRooms

Number of room types.

Section Reference: [Gamespy Peer SDK](#)

Presence SDK

Overview

The GameSpy Presence and Messaging SDK (GP) is an ANSI-C library that can be used by a game to add both account creation/authorization and "buddy list" functionality. If an application supports GP, its users can send messages back and forth with other users within that game, and with users in any other applications that use GP, such as GameSpy Arcade and other games. Through the use of a location string, users can see exactly what their online buddies are up to (in a game, in an Arcade staging room, reading news in Arcade, etc.). The location string also allows game specific information such as a server address - so if a user sees a buddy is online and playing the same game, he can just hop right onto the same server. In addition, GP allows users to get info on other users, including real name, e-mail address, ICQ UIN (user identification number), homepage, and zipcode. For privacy reasons, users can choose to hide some of this information from other users.

GP is purely data-based. The game is responsible for all graphical (or other) elements that allow a user to interact with it. There are no libraries or DLLs to deal with when using GP; just add the source files directly to your project and you're ready to go.

Two sample programs have been included:

- "gptest" is a Win32 MFC app that encapsulates all of GP's functionality in a single dialog box. This is not meant as a sample for how to do a UI for GP, but merely to show all of GP's capabilities in a single window. It's also useful for checking how a particular feature works.
- "gptestc" is a straight ANSI-C sample that connects a user, sends some messages to another user, then waits to receive some messages.

This document shows how to do some basic tasks with GP, such as connecting, creating a new account, and sending buddy messages. For more detailed information on GP, please see the GP reference

documentation.

File Manifest

The following files should be included with this package. If any of the files are missing, please [contact devsupport@gamespy.com](mailto:devsupport@gamespy.com).

File	Description
<i>gp.h</i>	GP header (all user functins are prototyped here)
<i>gp.c</i>	Entry point for all GP functions
<i>gpi.c</i> enabling/disabling	Code for intitialization/cleanup, processing and
<i>gpi.h</i>	Common header for internal code
<i>gpiBuddy.c,h</i>	Code for buddy messages
<i>gpiBuffer.c,h</i>	Code for socket buffering
<i>gpiCallback.c,h</i>	Code for adding/processing callbacks
<i>gpiConnect.c,h</i>	Code for connecting and disconnecting
<i>gpiInfo.c,h</i>	Code for getting and setting info
<i>gpiOperation.c,h</i>	Code for adding/removing/processing operations
<i>gpiPeer.c,h</i>	Code for direct peer-to-peer messaging
<i>gpiProfile.c,h</i>	Code for maintaining a list of profiles
<i>gpiSearch.c,h</i>	Code for dealing with search manager
<i>gpiUtility.c,h</i>	Miscellaneaous utility code
<i>nonport.c,h</i>	Platform-specific code
<i>md5c.c,md5.h</i>	MD5 code used for hashing

Implementation

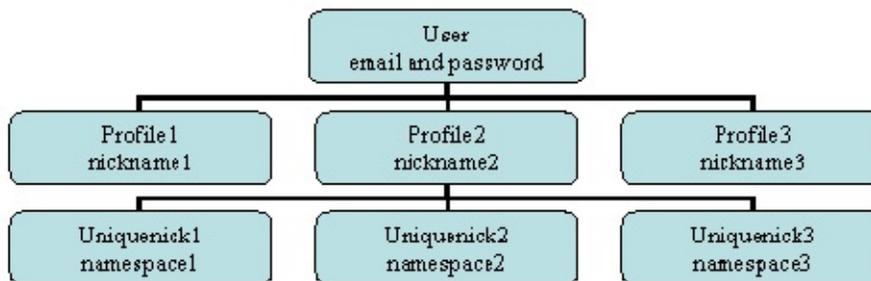
Accounts

When a user creates a new GP account, it is registered with an e-mail address, a nickname, and a password. The e-mail address must be unique among all users, because it identifies the particular "user". The nickname identifies the user's "profile". When an account is first created, it only has one profile. However, multiple profiles can be added. Each profile has a new nickname, but it is still associated with the same user, so the e-mail/password do not change. Multiple profiles belonging to a single user cannot have the same nickname, however profiles that belong to different users can have the same nickname.

When a user logs in, they login with a particular profile belonging to their user account. The e-mail address uniquely identifies the user, the password verifies the user, and the nickname uniquely identifies which of the user's profiles to use. A user cannot be logged in more than once simultaneously, even with different profiles. If a user attempts to login a user that is already logged in, the previous connection will be disconnected. This allows a user to login even if they forgot to log off on another computer.

Each profile can also have one or more unikenicks associated with it. A "uniquenick" is a special nickname that is unique in a given "namespace". There is a default GameSpy namespace which is used by GameSpy Arcade, but developers can also get their own namespaces. A profile can have one unikenick in each namespace. Because a unikenick can uniquely identify a profile in a given namespace, a unikenick and password combination can be used to login. However this is not recommended unless using a custom namespace with non-expiring unikenicks.

Otherwise once a unikenick expires, the user will no longer have a way of identifying that profile and will not be able to login.



The above chart shows the relationship between a user, its profiles, and any potential uniquenesses. In the top row is a user, which is identified by an email and password combination. Each user then has one or more profiles. A profile is identified by a nickname and the user to which it belongs. Each profile also has an associated profileid which can be used to identify it. Each profile can then have zero or more uniquenesses. Each uniqueness is identified by the profile to which it belongs and the namespace in which it exists. Because it is unique in its namespace, a uniqueness and namespace combination can be used to identify a profile (and in turn a user).

Types

There are a few basic types used by GP:

GPConnection

This is an object that represents an instance of GP. A pointer to a GP object is passed as the first argument to every GP function. For example:

```
GPConnection gp;
gpInitialize(&gp, productID, namespaceID, GP_PA
```

GPProfile

This is an object that represents a particular GP profile (either local or remote). A `GPProfile` object is passed to functions like `gpSendBuddyMessage` and `gpGetInfo`, and it is returned in

callbacks such as the [GP_RECV_BUDDY_STATUS](#) callback. A [GPProfile](#) object is equivalent to a profile ID. They are both int types, and can be used interchangeably.

GPCallback

This is a function type. Functions of this type are passed as parameters to [gpSetCallback](#) to set global (unsolicited) callbacks and to any functions that call a callback when completed (such as [gpConnect](#)). The first parameter is a pointer to this connection's [GPConnection](#) object, the second parameter is a pointer to a structure with callback-specific information, and the last parameter is a pointer to a user-supplied arg (which is passed as a parameter to the function to which the callback was passed). See *gp.h* for a list of all the arg structures.

```
typedef void (* GPCallback)(GPConnection * conn
```

GPResult

This is an enumeration of possible results from GP functions. A [GPResult](#) is returned from all GP functions (except for [gpDestroy](#) and [gpDisconnect](#), which have no return value). It is also passed as the first result in most callback arg structures. In args, it signals if there has been an error and, if so, what type of error.

[GP_NO_ERROR](#)

There has been no error.

[GP_MEMORY_ERROR](#)

A call to allocate memory failed.

[GP_PARAMETER_ERROR](#)

A parameter passed to a function was invalid.

[GP_NETWORK_ERROR](#)

There was an error reported by the underlying network layer.

[GP_SERVER_ERROR](#)

One of the backend servers returned an error.

GPEnum

[GPEnum](#) is an enumeration of various constants that are used as function parameters or are returned in callbacks.

Initializing

The first step in using GP is to initialize it with `gpInitialize`:

```
GPResult gpInitialize
(
    GPConnection * connection,
    int productID,
    int namespaceID,
    int partnerID
);
```

You need to pass it a pointer to a [GPConnection](#) object that you have declared or allocated. Typically, you will just declare a global [GPConnection](#) object and use that for all of your GP function calls. The [productID](#) is a unique ID that identifies your product. If you do not have a product ID, contact_devsupport@gamespy.com.

The [namespaceID](#) identified which namespace to login under. A [namespaceID](#) of 0 indicates that no namespace should be used. A [namespaceID](#) of 1 represents the default GameSpy namespace (the same namespace used by GameSpy Arcade). A [namespaceID](#) greater than 1 indicates a custom namespace. If uniqueness will not be used, [namespaceID](#) should be 0. Otherwise it should be 1, unless a custom namespace has been assigned.

The [partnerID](#) will typically be set to the value defined by `GP_PARTNERID_GAMESPY`.

If this call succeeds (returns [GP_NO_ERROR](#)), then GP is initialized and ready to be used. This instance of GP will be valid until [gpDestroy](#) is called with the same object.

After GP has been initialized, the next thing to do is set the global callbacks using [gpSetCallback](#):

```
GPResult gpSetCallback
(
    GPConnection * connection,
    GPEnum func,
    GPCallback callback,
    void * param
);
```

`func` is the type of callback, `callback` is the function to call for the callback, and `param` is a user-defined parameter that is passed to the callback. The possible values for `func` are:

GP_ERROR

This callback is called whenever a `GP_NETWORK_ERROR` or a `GP_SERVER_ERROR` occur. The arg passed to it is a `GPErrorArg`. The `errorCode` member of the arg can be checked for the specific cause of the error. If the "fatal" member of the arg is `GP_FATAL`, then an unrecoverable error has occurred, and the connection has already been disconnected, as if `gpDisconnect` were called. At this point, GP can be destroyed with `gpDestory`, or a new connection can be attempted with `gpConnect` (see below). If the "fatal" member of the arg is `GP_NON_FATAL`, then the user is still connected. At this point the application will likely show the user an error message (the "errorString" member of the `errorArg` can be used), and, optionally, ask the user to retry. The specific course of action can depend on the `errorCode`.

GP_RECV_BUDDY_REQUEST

This callback is called when another profile has made a request to add you to their buddy list.

GP_RECV_BUDDY_STATUS

This callback is called when there is updated status information for a buddy.

GP_RECV_BUDDY_MESSAGE

This callback is called when someone has sent you a buddy message.

GP_RECV_BUDDY_UTM

This callback is called when someone has sent you a UTM message.

[GP_RECV_GAME_INVITE](#)

This callback is called when someone invites you to play a particular game.

[GP_TRANSFER_CALLBACK](#)

This callback is called for status updates on a file transfer.

[GP_RECV_BUDDY_AUTH](#)

This callback is called when someone authorizes your buddy request.

[GP_RECV_BUDDY_REVOKE](#)

This callback is called when another profile revokes themselves as your buddy.

See the reference documentation for further details on each specific callback.

While GP is initialized, it must do some occasional processing to handle things like incoming buddy messages. `gpProcess` must be called by the application to allow for this processing. While it can be called as often as you like, it does not need to be called more than every second or so.

```
GPResult gpProcess
(
    GPConnection * connection
);
```

Connecting & Disconnecting

There are several functions that can be used to connect (login) to the Presence backend. `gpConnect` is used to login using a nick, email, and password. `gpConnectUniqueNick` allows you to connect using a uniquenick and password combination. `gpConnectPreAuthenticated` is used to connect using information from a partner authentication system.

```

GPResult gpConnect
(
    GPConnection * connection,
    const char nick[GP_NICK_LEN],
    const char email[GP_EMAIL_LEN],
    const char password[GP_PASSWORD_LEN],
    GPEnum firewall,
    GPEnum blocking,
    GPCallback callback,
    void * param
);

GPResult gpConnectUniqueNick
(
    GPConnection * connection,
    const char uniquenick[GP_UNIQUENICK_LEN],
    const char password[GP_PASSWORD_LEN],
    GPEnum firewall,
    GPEnum blocking,
    GPCallback callback,
    void * param
);

```

nick, uniquenick, email, password

identify the user account and the particular profile for that user.
nick, **uniquenick** and **e-mail** are not case-sensitive, however **password** is.

firewall

can be **GP_FIREWALL** or **GP_NO_FIREWALL**. If **GP_NO_FIREWALL**, then direct connections to other users will be attempted when sending buddy messages. If this is **GP_FIREWALL**, then all buddy messages will be sent and received through the server.

callback

will be called when the connection attempt is finished (successfully or not).

Nicknames

There are several possible ways to use these functions, depending on how the application plans on functioning.

GameSpyID Login with no uniquenick

Pass a 0 `namespaceID` to `gpInitialize`, which tells GP not to use namespaces.

When creating an account, use either `gpNewUser` or `gpConnectNewUser`, and set the `uniquenick` and `cdkey` parameters to `NULL`.

Call `gpConnect` to initiate the connection to the backend server.

GameSpyID Login with a uniquenick in the default namespace

Pass a `namespaceID` of 1 to `gpInitialize`, which identifies the default GameSpy namespace. This is the namespace that is used by GameSpy Arcade.

When creating an account, you'll want to call either `gpNewUser` or `gpConnectNewUser` and specify a `uniquenick` parameter. You can use this same value for the `nick` parameter as well.

To login to an account that has already been created, use the regular `gpConnect` function. Once the account has logged in, you can check the `uniquenick` member in the `GPConnectResponseArg` to see if there is a `uniquenick` associated with the profile. If the `uniquenick` is `@unregistered` or `@expired` then there is no `uniquenick` registered with this profile. In that case, use `gpRegisterUniqueNick` to assign a `uniquenick` to the profile. `gpSuggestUniqueNicks` can be used to get a list of `uniquenicks` to present to the user.

Make sure you do not use `gpConnectUniqueNick` for this method. The reason is that the default namespace expires `uniquenicks` after a certain period of inactivity, and you'll want to make sure the user can still login even if their `uniquenick` expired and was then taken by another user.

GameSpyID Login with a uniquenick in a custom namespace

Pass the namespace custom `namespaceID` to `gpInitialize`. Contact

devsupport@gamespy.com for information on obtaining a custom namespace.

If the namespace has expiring uniquenicks, then this method is identical to the above method, with the above exception of passing the custom namespaceID to gpInitialize.

If the namespace does not expire its uniquenicks, then the main difference between this and the above method is that you can use gpConnectUniqueNick to login. Because the uniquenick doesn't expire, a user only needs to remember his uniquenick and password to login. However it is still recommended that a valid email address be used when creating an account, as this will allow the user to retrieve a forgotten password.

Remote Authentication

The remote authentication login method is used to login using information from a partner authentication system. You login using a token and a challenge, which are supplied by the partner authentication system. Contact devsupport@gamespy.com for further information on using this login method.

When ready to log off the connection, use [gpDisconnect](#).

Creating & Deleting Profiles

To add a new profile to an existing account, use [gpNewProfile](#):

```
GPResult gpNewProfile(  
    GPConnection * connection,  
    const char nick[GP_NICK_LEN],  
    GPEnum replace,  
    GPEnum blocking,  
    GPCallback callback,  
    void * param  
);
```

[nick](#)

the nickname for the new profile.

replace

determines what should happen if the account already has a profile with the same nickname as the new one. Normally, this should be set to `GP_DONT_REPLACE`.

If there is an existing nickname, an error will be generated, with the `errorCode` set to `GP_NEWPROFILE_BAD_OLD_NICK` (the last `errorCode` generated can be checked with `gpGetErrorCode`). At this point, the user can be asked if he would like to replace the old profile. If he selects yes, then call `gpNewProfile` again, this time using `GP_REPLACE`.

The user must already be logged on with the account he wants to add the profile to when this function is called. To login under the new profile, the current profile must first be disconnected with `gpDisconnect`, then `gpConnect` called for the new one.

If the user would like to remove an unwanted profile, `gpDeleteProfile` can be used. It deletes the currently logged-in profile, so the user must connect with that profile before deleting it. As soon as `gpDeleteProfile` is called, the connection will be disconnected. GP will still be initialized, but the user must then login with a new profile to connect, or `gpDestroy` can be called to terminate GP. There is no way to delete an entire user account - if there is only one profile in an account, it cannot be deleted.

Searching

If the user wants to find a friend to add as a buddy, or just wants to find information on a certain person, they can search for the profile based on certain information. This is done using `gpProfileSearch`:

```
GPResult gpProfileSearch(  
    GPConnection * connection,  
    const char nick[GP_NICK_LEN],  
    const char email[GP_EMAIL_LEN],  
    const char firstname[GP_FIRSTNAME_LEN],  
    const char lastname[GP_LASTNAME_LEN],  
    int icquin,
```

```
GPEnum blocking,  
GPCallback callback,  
void * param  
);
```

Using the parameters above, the search can be based on nick, email, first name, last name, ICQ UIN, or any combination of the parameters. Pass in `NULL`, or an empty string, for any of the string parameters to ignore that parameter while searching. To ignore the ICQ UIN, pass in 0 for `icquin`.

Getting & Setting Info

To get information on a particular profile, use `gpGetInfo`:

```
GPRResult gpGetInfo(  
    GPCConnection * connection,  
    GPProfile profile,  
    GPEnum checkCache,  
    GPEnum blocking,  
    GPCallback callback,  
    void * param  
);
```

`profile`

the profile to get info on

`checkCache`

a flag that determines if the local cache should be checked for existing info on the profile. If it is `GP_CHECK_CACHE`, and the local cache has info on the profile, then that info will be used. If there is no locally cached info on the user, or if `GP_DONT_CHECK_CACHE` is used, then the info on the user will be retrieved from the backend server.

The info (either gotten locally or retrieved from the server) is passed to the callback in a `gpGetInfoResponseArg`:

```
typedef struct
{
    GPResult result;
    GPProfile profile;
    char nick[GP_NICK_LEN];
    char email[GP_EMAIL_LEN];
    char firstname[GP_FIRSTNAME_LEN];
    char lastname[GP_LASTNAME_LEN];
    char homepage[GP_HOMEPAGE_LEN];
    int icquin;
    char zipcode[GP_ZIPCODE_LEN];
    char countrycode[GP_COUNTRYCODE_LEN];
    int birthday;
    int birthmonth;
    int birthyear;
    GPEnum sex;
    GPEnum publicmask;
} GPGetInfoResponseArg;
```

The `gpSetInfo` group of functions are used to set information about the profile that is currently logged in. See the reference documentation for more info on all of the info that can be set. The user should be prompted to enter this information when a new account or new profile is created, and should have the option of changing the information at any later time.

Status

Every profile has a status, a status string, and a location string. This allows remote profiles to see if users on their buddy list are offline or online and, if online, what they are doing. There are currently six possible values for the status:

`GP_OFFLINE`

This profile is not connected to GP.

`GP_ONLINE`

This profile is online. This is the default status.

`GP_PLAYING`

This profile is playing a game.

GP_STAGING

This profile is in a staging room.

GP_CHATTING

This profile is chatting.

GP_AWAY

This profile is currently away from his computer (or the application).

Use `gpSetStatus` to set your status:

```
HRESULT gpSetStatus(  
    GPConnection * connection,  
    GPEnum status,  
    const char statusString[GP_STATUS_STRING_LEN],  
    const char locationString[GP_LOCATION_STRING_LEN]  
);
```

status

is one of the above values.

If `GP_OFFLINE` is set, it will cause the user to appear to be offline on buddy lists, but the GP connection will not actually be disconnected.

statusString

a user-readable description of the status. For example, if status is `GP_ONLINE`, this will typically be "Online".

locationString

a URL that describes the user's location. This can be used to, for example, join the staging room or server that a buddy is in.

When a user connects to GP, the initial status is `GP_ONLINE`, the initial status string is "Online", and the initial location string is an empty string. These do not change until `gpSetStatus` is called. The status will be reset again to the defaults the next time the user connects to GP.

When a user logs in, he is sent a status update for each of his buddies.

This is done through the `GP_RECV_BUDDY_STATUS` global callback, which is set with `gpSetCallback`. For example:

```
gpSetCallback(&gp, GP_RECV_BUDDY_STATUS, RecvBuddySt
```

Then, whenever a buddy's status changes, the callback is called again with the new information.

Buddies

In order to present the user with a buddy list, the application must do so by listening for buddy status messages (see above). After logging into GP, the full list of buddies will be available, though the status of each will not be known until the SDK receives a status update. Subsequent unsolicited messages after the login will update these buddy statuses asynchronously and trigger the `GP_RECV_BUDDY_STATUS` callback for each.

If the user wants to add a new buddy to their buddy list, `gpSendBuddyRequest` should be used.

```
GPResult gpSendBuddyRequest(  
    GPConnection * connection,  
    GPProfile profile,  
    const char reason[GP_REASON_LEN]  
);
```

This causes a message to be sent to the profile the user wants to add. This remote profile will receive a `GP_RECV_BUDDY_REQUEST` callback (again, registered with `gpSetCallback` as shown above). If the remote profile wants to authorize the request, it should call `gpAuthBuddyRequest`, passing in the `GPProfile` for the profile who made the request. The requesting profile will then receive a message letting him know that the request was authorized. And, if he's online, he will also receive a status update for the buddy. If the remote profile wants to deny the request, it should call `gpDenyBuddyRequest`.

Adding buddies is not reciprocal - in other words, adding a buddy to your

buddy list does not put you on that buddy's buddy list. Your new buddy still must make a request to add you to his buddy list, and you must authorize that in order for him to get you on his list.

To remove a buddy from your buddy list, use [gpDeleteBuddy](#):

```
GPResult gpDeleteBuddy(  
    GPConnection * connection,  
    GPProfile profile  
);
```

This will permanently delete the buddy from the buddy list. To get the buddy back on the list, a new request must be sent with [gpSendBuddyRequest](#). Also, if the buddy being removed has you on his buddy list, that will not be affected.

To send a message to a buddy, use [gpSendBuddyMessage](#):

```
GPResult gpSendBuddyMessage(  
    GPConnection * connection,  
    GPProfile profile,  
    const char * message  
);
```

This will send the given message to the profile, as long as it is actually an authorized buddy. If it is not, the call will fail ([errorCode](#) will be [GP_BM_NOT_BUDDY](#)). If [GP_FIREWALL](#) is set for either the local connection or the remote buddy, if the remote buddy is offline, or if a direct connection between buddies cannot be established for any other reason, the message will have to go through a backend server. If this happens, the message will be truncated. For this reason, buddy messages should not exceed about 4K bytes.

Blocked List

Similarly to a buddy list, GP also has the notion of a Blocked List. This list contains profiles with whom you want to block all GP communication to/from this player, including all form of messages, buddy requests, game

invites, etc. Essentially, it is as if this player is invisible to you. Note that the backend automatically handles blocking the traffic for members of the block list, so the game does not need to do anything extra to support this once a player has been added to the blocked list for a profile.

The Blocked List is retrieved in full upon a successful login. If you do plan on using this functionality in your game, it's recommended to have some sort of UI view that players can see the members of their block list in order to allow them to remove from it. Peer chat channel traffic still needs to be manually blocked via this list as it is not currently handled by the backend. To enumerate through the Blocked List, you can use [gpGetNumBlocked](#) in conjunction with [gpGetBlockedProfile](#). Examples of this are illustrated in the sample/test GP applications.

To add to the Blocked List, use [gpAddToBlockedList](#):

```
GPResult gpAddToBlockedList(  
    GPCConnection * connection,  
    GPProfile profile  
);
```

To remove from the Blocked List, use [gpRemoveFromBlockedList](#):

```
GPResult gpRemoveFromBlockedList(  
    GPCConnection * connection,  
    GPProfile profile  
);
```

Game Invitations

When considering game invitations, it is important to remember that invites are not technically restricted to players on the local buddy list. Any player may receive an invitation from any other player. Many developers will choose to implement their own design restrictions, such as limiting invites to clan or buddy list players, but this is not required.

Game invitations may be sent to a player using [gpInvitePlayer](#):

```
GPResult gpInvitePlayer
(
    GPConnection * connection,
    GPProfile profile,
    int productID,
    const gsi_char location[GP_LOCATION_STRING_L
);
```

The remote profile will receive a `GP_RECV_GAME_INVITE` callback for this invite request. The location parameter is an optional text string that usually contains the server IP and other connecting information. This parameter may be NULL. The max length for the location info is 255 characters. When compiling in Unicode mode, the location will be converted to ASCII.

An alternate design involves players inviting other players from a game lobby. In this case, the profileid of the remote player should be obtained from the Peer SDK [peerGetPlayerInfo](#).

Appendix I: Nickname Checks

There are various checks that are made on unquenicks before they can be registered.

Length

Uniquenicks must have at least 3 characters and no more than 20.

Validity

Alphanumeric characters (A...Z, a...z, 0...9). The first character may not be a digit.

"#\$%&()*+-./:;<=>?@[^_{}~ are allowed characters.

In ASCII codes the range is 34 to 126, excluding 44 (comma) and 92 (backslash) and 39 (apostrophe).

The 4 characters @+#: cannot be the first character in a nick.

Stripping

The uniqueness of a unquenick is determined based on it's "stripped" version. This is the unquenick with all non-alphanumeric characters removed. For example, "Joe", "%Joe%", and "Joe*" all have the same stripped version, "Joe".

This is a per-namespace option, and it is on in the default namespace.

Reserved Words

There is a per-namespace list of reserved words. If the stripped version of a unquenick matches a reserved word, then it cannot be registered. For example, "server" is a reserved word in the default namespace, so the unquenicks "server", "%server%", and "server*" would not be allowed.

Filtering

Uniquenicks are filtered on a per-namespace basis. Each namespace can have a list of patterns which all unquenicks are checked against before they are allowed to be registered. This is primarily used to prevent nicks with "bad words" in them.

Appendix II: PS3 Integration with NP

Remote Authentication - How to login with the PS3

GameSpy honors the Playstation Network (NP) single sign-on principle, by supporting NP handles via remote authentication. You would use the details for the current NP user you're logged in under and GameSpy's backend will provide you with the profile ID and connection handle to a 'shadow account', which acts just like any other GameSpy ID account for the rest of our API calls. Remote authentication is as easy as following these steps:

- Request and obtain your NP ticket from Sony using your unique NP ID. This is done by using the [PS3 NP Manager lib](#) and calling [sceNpManagerRequestTicket](#) followed by [sceNpManagerGetTicket](#).
- Use the GameSpy AuthService (webservices folder) to convert the NP ticket into a remote auth token. This is accomplished by calling [wsLoginPs3Cert](#). This requires you to specify a partnercode/namespaceID which is specific to the PS3. They are:

(Live) PS3 PartnerCode (or partnerid): 19

(Live) PS3 NamespaceID: 28

Note that the namespaceID/partnercode listed above applies to Live PS3 accounts registered in the 'NP' environment. If you are using Development accounts ('sp-int' environment) you will want to use the development namespaceID/partnercode otherwise the login will fail. For sp-int accounts, the namespaceID/partnercode pair to use instead is:

(Dev) PS3 PartnerCode: 33

(Dev) PS3 NamespaceID: 40

- Once you have the authtoken/partnerchallenge from the callback that returns from the above call, these values are used with our remote authentication systems to log the player into the GameSpy backend like you would normally with GameSpy ID. For example, to

login to GP you would call `gpConnectPreAuthenticated` passing in the retrieved `authToken/partnerChallenge` from the `AuthService` callback (initializing GP with the PS3 namespaceid and partnercode given above).

- If integrating with our ATLAS SDK, you would use the `AuthService` function `wsLoginRemoteAuth` (using the same namespaceID and partnerCode as the login Ps3Cert call) to log the player into the backend and retrieve the necessary Certificate/PrivateData for ATLAS.

The shadow accounts created for these users are in their own unique namespace for the NP system and their `uniquenick == NP ID`, so you will be able to use your PS3 account name as your `uniquenick`.

Note that the PS3 `AuthService` requires a cipher file in order to authenticate NP tickets for your title. This cipher is tied to your service ID used in NP and is submitted to us by Sony. You can start a support ticket with Sony to get this process started if you're experiencing authentication problems.

GP-NP Buddy + Block List Synchronization

By default on the PlayStation 3 platform, GP integrates with the NP system in order to seamlessly sync a PS3 user's NP Buddy list & Block list into their respective GP account. This sync effectively checks to see if any of your NP Buddies or Blocks have created GP 'shadow accounts' (e.g. have played a GameSpy-enabled PS3 title) and if so, will add them to your GP account in order to show up in-game.

This PS3 Buddy & Block sync takes place immediately after login. The SDK will first try to initialize NP Basic and NP Lookup - if either have already been initialized this is perfectly fine, the SDK will leave them intact and not destroy them upon calling `gpDestroy` assuming the game will take care of this. After initializing NP, the SDK waits a short period of time (`GPI_NP_SYNC_DELAY`) in order to allow NP Basic to acquire the buddy and block lists. After this delay, the actual sync takes place; note that since the SDK needs to verify if the NP players have valid GP accounts, `gpProcess` should be called routinely after the login in order to allow processing to take place (the sync is asynchronous).

To monitor the sync progress you can view the debugging commentary by defining [GSI_COMMON_DEBUG](#) and set the debug level appropriately (verbose tells you everything).

Keep in mind that the **namespaceid** and **partnerid** used by GP should correspond to the NP environment being utilized on the PS3, otherwise the sync will not work as intended. You can reference the above *Remote Authentication* section for more information about which identifiers to use for the respective NP environment.

In addition to the initial sync, GP also supports NP by mirroring requests to add players to their GP Blocked List. When you add to your blocked list on the PS3 using [gpAddToBlockedList](#), the SDK will attempt to mirror this addition to the NP Block list as well. The SDK will perform an NP lookup to see if the player exists in the NP environment, and if so, then will try to add them to the NP block list. Please note that this process is also entirely asynchronous and dependent upon routinely calling [gpProcess](#) to allow the SDK to continue it's processing.

Presence and Messaging SDK Functions

gpAcceptTransfer	This function is used to accept a file transfer request.
gpAddToBlockedList	Adds a remote profile to the local player's blocked list.
gpAuthBuddyRequest	This function authorizes a buddy request. It is called in response to the gpRecvBuddyRequest callback getting called.
gpCheckUser	Validates a user's info, without logging into the account.
gpConnect	This function is used to establish a connection to the server. It establishes a connection with an existing profile, which is identified based on the nick and email and is validated by the password.
gpConnectNewUser	This function is used to create a new user account and profile and to then establish a connection using the profile.
gpConnectPreAuthenticated	This function is used to establish a connection to the server. It establishes a connection using an authtoken and a partnerchallenge, both obtained from a partner authentication system.

gpConnectUniqueNick	This function is used to establish a connection to the server. It establishes a connection with an existing profile, which is identified based on the uniquenick and is validated by the password.
gpDeleteBuddy	This function deletes a buddy from the local profile's buddy list.
gpDeleteProfile	This function deletes the local profile. Note that this is a blocking call.
gpDenyBuddyRequest	This function denies a buddy request. It is called in response to the gpRecvBuddyRequest callback getting called.
gpDestroy	This function is used to destroy a connection object.
gpDisable	This function disables a certain state.
gpDisconnect	This function terminates the local connection. This should always be called when the connection is no longer needed.
gpEnable	This function enables a certain state.
gpFreeTransfer	This function is used to free a file transfer.
gpGetBlockedProfile	

	This function gets the profileid for a particular player on the blocked list.
gpGetBuddyIndex	This function checks a remote profile to see if it is a buddy. If it is a buddy, the buddy's index is returned. If it is not a buddy, the index will be set to -1
gpGetBuddyStatus	This function gets the status for a particular buddy on the buddy list.
gpGetCurrentFile	This function is used to get the current file being transferred.
gpGetErrorCode	This function gets the current error code for a connection.
gpGetErrorString	This function gets the current error string for a connection.
gpGetFileModificationTime	This function is used to get a file's timestamp.
gpGetFileName	This function is used to get the name of a file.
gpGetFilePath	This function is used to get the local path to a file.
gpGetFileProgress	This function is used to get the progress of a file being transferred.

gpGetFileSize	This function is used to get the size of a file being transferred.
gpGetInfo	This function gets info on a particular profile.
gpGetLoginTicket	Retrieves a connection "token" that may be used by HTTP requests to uniquely identify the player.
gpGetNumBlocked	Gets the total number of blocked players in the local profile's blocked list.
gpGetNumBuddies	This function gets the number of buddies on the local profile's buddy list.
gpGetNumFiles	This function is used to get the number of files (including directories) being transferred.
gpGetNumTransfers	Returns the number of pending file transfers.
gpGetReverseBuddies	Get profiles that have you on their buddy list.
gpGetTransfer	Returns the GPTransfer object at the specified index.
gpGetTransferData	

	This function is used to retrieve arbitrary user-data stored with a transfer.
gpGetTransferProfile	This function is used to get the remote profile for a transfer.
gpGetTransferProgress	This function is used to get the total progress of the transfer, in bytes.
gpGetTransferSide	This function is used to get which side of the transfer the local profile is on (sending or receiving).
gpGetTransferSize	This function is used to get the total size of the transfer, in bytes.
gpGetTransferThrottle	This function can be used to get a transfer's throttle setting. NOTE: Throttling is not currently implemented. Throttle information is transmitted between the local profile and remote profile, but no throttling actually occurs.
gpGetUserNicks	This function gets the nicknames for a given e-mail/password (which identifies a user).
gpIDFromProfile	A GPPProfile is now the same as a profileid.
gpInitialize	This function is used to initialize a connection object.

gpInvitePlayer	This function invites a player to play a certain game.
gplsBlocked	Returns gsi_true if the given ProfileID is blocked, gsi_false if not blocked.
gplsBuddy	Returns 1 if the given ProfileID is a buddy, 0 if not a buddy
gplsConnected	Determine whether the GPConnection object has established a connection with the server.
gplsValidEmail	This function checks if there is an account with the given e-mail address.
gpNewProfile	This function creates a new profile for the local user.
gpNewUser	This function creates a new user account and a profile in that account. Unlike gpConnectNewUser, gpNewUser does not login with the new account. The local user does not need to be connected to use this function.
gpProcess	This function does any necessary processing that needs to be done on connection.
gpProfileFromID	Translates a profile id into a GPProfile.

gpProfileSearch	This function searches for profiles based on certain criteria.
gpProfilesReport	Debug function to dump information on known profiles to the console.
gpRegisterCdKey	This function attempts to register a cdkey and associate it with the local profile.
gpRegisterUniqueNick	This function attempts to register a uniquenick and associate it with the local profile.
gpRejectTransfer	This function is used to reject a file transfer request.
gpRemoveFromBlockedList	Removes a remote profile from the local player's blocked list.
gpRevokeBuddyAuthorization	Remove the local client from a remote users buddy list.
gpSendBuddyMessage	This function sends a message to a buddy.
gpSendBuddyRequest	This function sends a request to a remote profile to ask for permission to add the remote profile to the local profile's buddy list.

gpSendBuddyUTM	Sends a UTM (under-the-table) message to a buddy.
gpSendFiles	This function attempts to send one or more files (and/or sub-directory names) to another profile.
gpSetCallback	This function is used to set callbacks. The callbacks that get set with this function are called as a result of data received from the server, such as messages or status updates.
gpSetInfoCacheFilename	Sets the file name for the internal profile cache.
gpSetInfod	These functions are used to set local info.
gpSetInfoi	These functions are used to set local info.
gpSetInfoMask	
gpSetInfos	These functions are used to set local info.
gpSetStatus	This function sets the local profile's status.
gpSetTransferData	This function is used to store arbitrary user-data with a transfer.
gpSetTransferDirectory	This function can be used to set the directory that files are received into.

[gpSetTransferThrottle](#)

This function can be used to set a throttle on a transfer. NOTE: Throttling is not currently implemented. Throttle information is transmitted between the local profile and remote profile, but no throttling actually occurs.

[gpSkipFile](#)

This function is used to skip transferring a certain file.

[gpSuggestUniqueNick](#)

This function gets suggested uniqueness from the backend.

[gpUserIDFromProfile](#)

This function gets a profile's user ID.

gpAcceptTransfer

This function is used to accept a file transfer request.

```
GPRResult gpAcceptTransfer(  
    GPConnection * connection,  
    GPTransfer transfer,  
    const gsi_char * message );
```

Routine	Required Header	Distribution
gpAcceptTransfer	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] The connection on which to accept the transfer.

transfer

[in] The transfer passed along with the
GP_TRANSFER_SEND_REQUEST.

message

[in] An optional message to send along with the accept.

Remarks

This function is used to accept an incoming files request. This will initiate the transfer from the remote profile to the local profile. When done with the transfer, the transfer should be freed with a call to `gpFreeTransfer`.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
gpAcceptTransfer	gpAcceptTransferA	gpAcceptTransferW

gpAcceptTransferW and **gpAcceptTransferA** are UNICODE and ANSI mapped versions of **gpAcceptTransfer**. The arguments of **gpAcceptTransferA** are ANSI strings; those of **gpAcceptTransferW** are wide-character strings.

Section Reference: [Gamespy Presence SDK](#)

See Also: [gpRejectTransfer](#), [gpSendFiles](#)

gpAddToBlockedList

Adds a remote profile to the local player's blocked list.

```
GPRResult gpAddToBlockedList(  
    GPConnection * connection,  
    GPProfile profile );
```

Routine	Required Header	Distribution
gpAddToBlockedList	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] A GP connection interface.

profile

[in] The profileid of the player to be blocked.

Remarks

A blocked player is essentially invisible to player who has him/her blocked. The local player will not receive any communication from the blocked player, nor will the local player be able to contact the blocked player in any way.

This function will only work when GP is connected. This function will not return any callback upon success, but the GP_ERROR callback will be called should an error occur during the add attempt.

Section Reference: [Gamespy Presence SDK](#)

See Also: [gpRemoveFromBlockedList](#), [gpGetNumBlocked](#), [gpGetBlockedProfile](#), [gplsBlocked](#)

gpAuthBuddyRequest

This function authorizes a buddy request. It is called in response to the gpRecvBuddyRequest callback getting called.

```
GPRResult gpAuthBuddyRequest(  
    GPConnection * connection,  
    GPProfile profile );
```

Routine	Required Header	Distribution
gpAuthBuddyRequest	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] The connection on which to authorize the request.

profile

[in] The remote profile whose buddy request is being authorized.

Remarks

This function is used to authorize a buddy request received with the gpRecvBuddyRequest callback. It is used only to authorize. This function does not need to be called immediately after a request has been received, however the request will be lost as soon as the local profile is disconnected.

This function causes a status message to be sent to the remote profile.

Section Reference: [Gamespy Presence SDK](#)

gpCheckUser

Validates a user's info, without logging into the account.

```
GPResult gpCheckUser(  
    GPConnection * connection,  
    const gsi_char nick[GP_NICK_LEN],  
    const gsi_char email[GP_EMAIL_LEN],  
    const gsi_char password[GP_PASSWORD_LEN],  
    GPEnum blocking,  
    GPCallback callback,  
    void * param );
```

Routine	Required Header	Distribution
gpCheckUser	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] A GP interface object initialized with `gpInitialize`. (Does not have to be connected.)

nick

[in] The profile nickname.

email

[in] The profile email address.

password

[in] The profile password.

blocking

[in] `GP_BLOCKING` or `GP_NON_BLOCKING`

callback

[in] A user supplied callback with an arg type of `GPConnectResponseArg`.

param

[in] Pointer to user defined data. This value will be passed unmodified to the callback function.

Remarks

This function is rarely used but may be useful in certain situations. The main advantage is that a user's info may be verified without disrupting other external connections. (gpConnect will usurp any previous connections).

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
gpCheckUser	gpCheckUserA	gpCheckUserW

gpCheckUserW and **gpCheckUserA** are UNICODE and ANSI mapped versions of **gpCheckUser**. The arguments of **gpCheckUserA** are ANSI strings; those of **gpCheckUserW** are wide-character strings.

Section Reference: [Gamespy Presence SDK](#)

See Also: [GPCheckResponseArg](#)

gpConnect

This function is used to establish a connection to the server. It establishes a connection with an existing profile, which is identified based on the nick and email and is validated by the password.

```
GPResult gpConnect(  
    GPConnection * connection,  
    const gsi_char nick[GP_NICK_LEN],  
    const gsi_char email[GP_EMAIL_LEN],  
    const gsi_char password[GP_PASSWORD_LEN],  
    GPEnum firewall,  
    GPEnum blocking,  
    GPCallback callback,  
    void * param );
```

Routine	Required Header	Distribution
gpConnect	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] A GP interface object initialized with `gpInitialize`.

nick

[in] The profile nickname.

email

[in] The profile email address.

password

[in] The profile password.

firewall

[in] `GP_FIREWALL` or `GP_NO_FIREWALL`. This option may limit the users ability to transfer files.

blocking

[in] `GP_BLOCKING` or `GP_NON_BLOCKING`

callback

[in] A user-supplied callback with an arg type of `GPConnectResponseArg`

param

[in] Pointer to user-defined data. This value will be passed unmodified to the callback function.

Remarks

This function establishes a connection with the server. If the local machine is behind a firewall, the firewall parameter should be set to GP_FIREWALL so that buddy messages can be sent through the server. gpDisconnect should be called when this connection is ready to be disconnected..

When the connection is complete, the callback will be called.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
gpConnect	gpConnectA	gpConnectW

gpConnectW and **gpConnectA** are UNICODE and ANSI mapped versions of **gpConnect**. The arguments of **gpConnectA** are ANSI strings; those of **gpConnectW** are wide-character strings.

Section Reference: [Gamespy Presence SDK](#)

See Also: [GPConnectResponseArg](#)

gpConnectNewUser

This function is used to create a new user account and profile and to then establish a connection using the profile.

```
GPResult gpConnectNewUser(  
    GPConnection * connection,  
    const gsi_char nick[GP_NICK_LEN],  
    const gsi_char uniquenick[GP_UNIQENICK_LEN],  
    const gsi_char email[GP_EMAIL_LEN],  
    const gsi_char password[GP_PASSWORD_LEN],  
    const gsi_char cdkey[GP_CDKEY_LEN],  
    GPEnum firewall,  
    GPEnum blocking,  
    GPCallback callback,  
    void * param );
```

Routine	Required Header	Distribution
gpConnectNewUser	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] A GP interface object initialized with gpInitialize.

nick

[in] The desired nickname for the initial profile. The nickname can be up to GP_NICK_LEN characters long, including the NULL.

uniquenick

[in] The desired unique nickname for the profile.

email

[in] The desired e-mail address for the user. Can be up to GP_EMAIL_LEN characters long, including the NUL terminator.

password

[in] The desired password for the profile. The password can be up to GP_PASSWORD_LEN characters long, including the NUL.

cdkey

[in] An optional cdkey to associate with the unique nick. Normally left blank.

firewall

[in] GP_FIREWALL or GO_NO_FIREWALL. This option may limit the users ability to send files.

blocking

[in] GP_BLOCKING or GP_NON_BLOCKING

callback

[in] A user supplied callback with an arg type of GPConnectResponseArg

param

[in] Pointer to user defined data. This value will be passed unmodified to the callback function.

Remarks

This function is identical to gpConnect (see above), except that it first creates a new user and profile, and then connects the profile. If this function is used to try to connect a profile that already exists, the operation will fail. If the e-mail and password identify an existing user, but the nick does not match any of that user's profiles, a new profile will be created and logged in.

If using uniquenicks, then you will normally want to use the same string for both the nick and uniquenick parameters. If namespaceID is 0 then the uniquenick and cdkey parameters should be NULL.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
gpConnectNewUser	gpConnectNewUserA	gpConnectNewUserW

gpConnectNewUserW and **gpConnectNewUserA** are UNICODE and ANSI mapped versions of **gpConnectNewUser**. The arguments of **gpConnectNewUserA** are ANSI strings; those of **gpConnectNewUserW** are wide-character strings.

Section Reference: [Gamespy Presence SDK](#)

See Also: [GPCConnectResponseArg](#)

gpConnectPreAuthenticated

This function is used to establish a connection to the server. It establishes a connection using an authtoken and a partnerchallenge, both obtained from a partner authentication system.

```
GPResult gpConnectPreAuthenticated(  
    GPConnection * connection,  
    const gsi_char authtoken[GP_AUTHTOKEN_LEN],  
    const gsi_char  
    partnerchallenge[GP_PARTNERCHALLENGE_LEN],  
    GPEnum firewall,  
    GPEnum blocking,  
    GPCallback callback,  
    void * param );
```

Routine	Required Header	Distribution
gpConnectPreAuthenticated	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] A GP interface object initialized with `gpInitialize`.

authtoken

[in] An authentication token generated by a partner database.

partnerchallenge

[in] The challenge received from the partner database.

firewall

[in] `GP_FIREWALL` or `GO_NO_FIREWALL`. This option may limit the users ability to send files.

blocking

[in] `GP_BLOCKING` or `GP_NON_BLOCKING`

callback

[in] A user-supplied callback with an arg type of `GPConnectResponseArg`

param

[in] Pointer to user-defined data. This value will be passed unmodified to the callback function.

Remarks

This function establishes a connection with the server. If the local machine is behind a firewall, the firewall parameter should be set to GP_FIREWALL so that buddy messages can be sent through the server. gpDisconnect should be called when this connection is ready to be disconnected..

When the connection is complete, the callback will be called.

This function should only be used if the namespaceID parameter passed to gpInitialize was greater than 0.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICOD
gpConnectPreAuthenticated	gpConnectPreAuthenticatedA	gpConnectPre

gpConnectPreAuthenticatedW and **gpConnectPreAuthenticatedA** are UNICODE and ANSI mapped versions of **gpConnectPreAuthenticated**. The arguments of **gpConnectPreAuthenticatedA** are ANSI strings; those of **gpConnectPreAuthenticatedW** are wide-character strings.

Section Reference: [Gamespy Presence SDK](#)

See Also: [GPConnectResponseArg](#)

gpConnectUniqueNick

This function is used to establish a connection to the server. It establishes a connection with an existing profile, which is identified based on the uniquenick and is validated by the password.

```
GPResult gpConnectUniqueNick(  
    GPConnection * connection,  
    const gsi_char uniquenick[GP_UNIQUE_NICK_LEN],  
    const gsi_char password[GP_PASSWORD_LEN],  
    GPEnum firewall,  
    GPEnum blocking,  
    GPCallback callback,  
    void * param );
```

Routine	Required Header	Distribution
gpConnectUniqueNick	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] A GP interface object initialized with `gpInitialize`.

uniquenick

[in] The `uniquenick`.

password

[in] The profile password.

firewall

[in] `GP_FIREWALL` or `GO_NO_FIREWALL`. This option may limit the users ability to send files.

blocking

[in] `GP_BLOCKING` or `GP_NON_BLOCKING`

callback

[in] A user-supplied callback with an arg type of `GPConnectResponseArg`

param

[in] Pointer to user-defined data. This value will be passed unmodified to the callback function.

Remarks

This function establishes a connection with the server. If the local machine is behind a firewall, the firewall parameter should be set to GP_FIREWALL so that buddy messages can be sent through the server. gpDisconnect should be called when this connection is ready to be disconnected..

When the connection is complete, the callback will be called.

This function should only be used in a custom namespace that does not expire uniqueness. This is because if a uniqueness expires and is then taken by another user, the original user will no longer be able to login using that uniqueness.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
gpConnectUniqueNick	gpConnectUniqueNickA	gpConnectUniqueNickW

gpConnectUniqueNickW and **gpConnectUniqueNickA** are UNICODE and ANSI mapped versions of **gpConnectUniqueNick**. The arguments of **gpConnectUniqueNickA** are ANSI strings; those of **gpConnectUniqueNickW** are wide-character strings.

Section Reference: [Gamespy Presence SDK](#)

See Also: [GPConnectResponseArg](#)

gpDeleteBuddy

This function deletes a buddy from the local profile's buddy list.

```
GPRResult gpDeleteBuddy(  
    GPCConnection * connection,  
    GPProfile profile );
```

Routine	Required Header	Distribution
gpDeleteBuddy	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] A GP connection interface with an established connection.

profile

[in] The profile ID of the buddy to delete.

Remarks

This function deletes the buddy indicated by profile from the local profile's buddy list.

Section Reference: [Gamespy Presence SDK](#)

gpDeleteProfile

This function deletes the local profile. Note that this is a blocking call.

```
GPRResult gpDeleteProfile(  
    GPConnection * connection,  
    GPCallback callback,  
    void * arg );
```

Routine	Required Header	Distribution
gpDeleteProfile	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] A GP connection interface with an established connection.

callback

[in] The callback used to confirm the deleted profile

arg

[in] User data

Remarks

This function deletes the local profile. Because the connection is between the local profile and the server, this automatically ends this connection (gpDisconnect does not need to be called). There is no way to delete any profile other than the current connected profile. The operation will fail if the connected profile is the user's only profile. A successful delete will result in the callback getting called. The callback will have the data about the delete profile and whether it was successful or not.

Section Reference: [Gamespy Presence SDK](#)

gpDenyBuddyRequest

This function denies a buddy request. It is called in response to the gpRecvBuddyRequest callback getting called.

```
GPRResult gpDenyBuddyRequest(  
    GPConnection * connection,  
    GPProfile profile );
```

Routine	Required Header	Distribution
gpDenyBuddyRequest	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] A GP connection interface with an established connection.

profile

[in] The profile ID of the player who sent the AddBuddyRequest; i.e., the player you are denying

Remarks

This function is used to deny a buddy request received with the gpRecvBuddyRequest callback. This function does not need to be called immediately after a request has been received. Nothing is sent to the remote profile letting them know the request was denied.

Section Reference: [Gamespy Presence SDK](#)

gpDestroy

This function is used to destroy a connection object.

```
void gpDestroy(  
    GPCConnection * connection );
```

Routine	Required Header	Distribution
gpDestroy	<gp.h>	SDKZIP

Parameters

connection

[in] A GP connection interface.

Remarks

This function destroys a connection object. This should be called when a GPConnection object is no longer needed. The object cannot be used after it has been destroyed.

Section Reference: [Gamespy Presence SDK](#)

gpDisable

This function disables a certain state.

```
GPRresult gpDisable(  
    GPCConnection * connection,  
    GPEnum state );
```

Routine	Required Header	Distribution
gpDisable	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] A GP connection interface.

state

[in] The "state" to disable.

Remarks

This function is used to disable ("turn off") states on the connection. The states that can currently be disabled are info caching and simulation. To enable a state use gpEnable.

Section Reference: [Gamespy Presence SDK](#)

gpDisconnect

This function terminates the local connection. This should always be called when the connection is no longer needed.

```
void gpDisconnect(  
    GPCConnection * connection );
```

Routine	Required Header	Distribution
gpDisconnect	<gp.h>	SDKZIP

Parameters

connection

[in] A GP connection interface.

Remarks

This function should be called to disconnect a connection when it is no longer needed. After this call, connection can be reused for a new connection.

Section Reference: [Gamespy Presence SDK](#)

See Also: [gpDestroy](#)

gpEnable

This function enables a certain state.

```
GPRResult gpEnable(  
    GPConnection * connection,  
    GPEnum state );
```

Routine	Required Header	Distribution
gpEnable	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] A GP connection interface.

state

[in] The "state" to enable.

Remarks

This function is used to enable ("turn on") states on the connection. The states that can currently be enabled are info caching and simulation. To disable a state use gpDisable.

Section Reference: [Gamespy Presence SDK](#)

gpFreeTransfer

This function is used to free a file transfer.

```
GPRresult gpFreeTransfer(  
    GPCConnection * connection,  
    GPTransfer transfer );
```

Routine	Required Header	Distribution
gpFreeTransfer	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] A GP connection interface.

transfer

[in] A pointer to a GPTransfer object.

Remarks

This function is used to free a transfer object. If the transfer has completed, then this will simply free the object's resources. If the transfer has not yet completed, this will also cancel the transfer, causing the remote profile to get a GP_TRANSFER_CANCELLED callback.

Section Reference: [Gamespy Presence SDK](#)

See Also: [gpSendFiles](#), [gpAcceptTransfer](#), [gpRejectTransfer](#)

gpGetBlockedProfile

This function gets the profileid for a particular player on the blocked list.

```
GPRResult gpGetBlockedProfile(  
    GPCConnection * connection,  
    int index,  
    GPProfile * profile );
```

Routine	Required Header	Distribution
gpGetBlockedProfile	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] A GP connection interface.

index

[in] The array index of the blocked player.

profile

[out] The profileid of the blocked player.

Remarks

The blocked list is fully obtained after the login process is complete. Index is a number greater than or equal to 0 and less than the total number of blocked players - generally called in conjunction with `gpGetNumBlocked` to enumerate through the list.

Section Reference: [Gamespy Presence SDK](#)

See Also: [gpGetNumBlocked](#), [gplsBlocked](#)

gpGetBuddyIndex

This function checks a remote profile to see if it is a buddy. If it is a buddy, the buddy's index is returned. If it is not a buddy, the index will be set to -1.

```
GPRResult gpGetBuddyIndex(  
    GPConnection * connection,  
    GPProfile profile,  
    int * index );
```

Routine	Required Header	Distribution
gpGetBuddyIndex	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] A GP connection interface.

profile

[in] The profile ID of the buddy.

index

[out] The internal array index of the buddy.

Remarks

This function is used to check if a remote profile is a buddy and to get its buddy index if it is a buddy. This buddy index can then be used in a call to `gpGetBuddyStatus`.

The buddy index may become invalid after a buddy is added to or deleted from the buddy list. If the profile is not a buddy, `GP_NO_ERROR` will be returned (as long as no other errors happen), and index will be set to -1.

Section Reference: [Gamespy Presence SDK](#)

gpGetBuddyStatus

This function gets the status for a particular buddy on the buddy list.

```
GPResult gpGetBuddyStatus(  
    GPConnection * connection,  
    int index,  
    GPBuddyStatus * status );
```

Routine	Required Header	Distribution
gpGetBuddyStatus	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] A GP connection interface.

index

[in] The array index of the buddy.

status

[out] The status of this buddy.

Remarks

This function is used to get the status of a particular buddy. index is a number greater than or equal to 0 and less than the total number of buddies.

This function will typically be called in response to the gpRecvBuddyStatus callback being called.

Section Reference: [Gamespy Presence SDK](#)

gpGetCurrentFile

This function is used to get the current file being transferred.

```
GPResult gpGetCurrentFile(  
    GPConnection * connection,  
    GPTransfer transfer,  
    int * index );
```

Routine	Required Header	Distribution
gpGetCurrentFile	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] A GP connection interface.

transfer

[in] A GP transfer object

index

[out] Returns the index of the current transferring file.

Remarks

This function is used to get the index of the current file being transferred. This will be 0 until the first file is finished, then 1 until the second file finishes, etc. When the transfer is complete, it will be set to the number of files in the transfer.

Section Reference: [Gamespy Presence SDK](#)

gpGetErrorCode

This function gets the current error code for a connection.

```
GPRResult gpGetErrorCode(  
    GPConnection * connection,  
    GPErrCode * errorCode );
```

Routine	Required Header	Distribution
gpGetErrorCode	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] A GP connection interface.

errorCode

[out] The current error code.

Remarks

This function gets the current error code for connection. It can be used to determine the specific cause of the most recent error. See the GP header, gp.h, for all of the possible error codes.

Section Reference: [Gamespy Presence SDK](#)

gpGetErrorString

This function gets the current error string for a connection.

```
GPRResult gpGetErrorString(  
    GPConnection * connection,  
    gsi_char errorString[GP_ERROR_STRING_LEN] );
```

Routine	Required Header	Distribution
gpGetErrorString	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] A GP connection interface.

errorString

[in] A text description of the current error.

Remarks

This function gets the current error string for connection. The error string is a text description of the most recent error that occurred on this connection. If no errors have occurred on this connection, the error string will be empty ("").

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
gpGetErrorString	gpGetErrorStringA	gpGetErrorStringW

gpGetErrorStringW and **gpGetErrorStringA** are UNICODE and ANSI mapped versions of **gpGetErrorString**. The arguments of **gpGetErrorStringA** are ANSI strings; those of **gpGetErrorStringW** are wide-character strings.

Section Reference: [Gamespy Presence SDK](#)

gpGetFileModificationTime

This function is used to get a file's timestamp.

```
GPRResult gpGetFileModificationTime(  
    GPConnection * connection,  
    GPTransfer transfer,  
    int index,  
    unsigned long * modTime );
```

Routine	Required Header	Distribution
gpGetFileModificationTime	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] A GP connection interface.

transfer

[in] A pointer to a GPTransfer object.

index

[in] Index of the file within the GPTransfer object.

modTime

[out] The modification time.

Remarks

This function is used to get the timestamp for a file being transferred. This is typically used by the receiver to set the file's timestamp correctly after a file has been received.

Section Reference: [Gamespy Presence SDK](#)

gpGetFileName

This function is used to get the name of a file.

```
GPRResult gpGetFileName(  
    GPConnection * connection,  
    GPTransfer transfer,  
    int index,  
    gsi_char ** name );
```

Routine	Required Header	Distribution
gpGetFileName	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] A GP connection interface.

transfer

[in] A pointer to a GPTransfer object.

index

[in] The index of the file within the GPTransfer object.

name

[out] The name of the file.

Remarks

This function is used to get the name of a file in the transfer. The receiver should use this name to determine where to put the file after it is received. It may be a simple name ("file.ext"), or it may contain a directory path ("files/file.ext"). Any slashes in the name will be UNIX-style slashes ("files/file.ext") as opposed to Windows style slashes ("files\file.ext").

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
gpGetFileName	gpGetFileNameA	gpGetFileNameW

gpGetFileNameW and **gpGetFileNameA** are UNICODE and ANSI mapped versions of **gpGetFileName**. The arguments of **gpGetFileNameA** are ANSI strings; those of **gpGetFileNameW** are wide-character strings.

Section Reference: [Gamespy Presence SDK](#)

gpGetFilePath

This function is used to get the local path to a file.

```
GPRresult gpGetFilePath(  
    GPConnection * connection,  
    GPTransfer transfer,  
    int index,  
    gsi_char ** path );
```

Routine	Required Header	Distribution
gpGetFilePath	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] A GP connection interface.

transfer

[in] A pointer to a GPTransfer object.

index

[in] The index of the file within the GPTransfer object.

path

[in] The path of the file.

Remarks

This function is used to get the local path to a file. For the sender, this will be the same path specified in the `gpSendFilesCallback`. For the receiver, this will be `NULL` for directories and for files that haven't started transferring yet. For files that have are either transferring or have finished transferring, this is the local path where the file is being stored. It is the application's responsibility to move the file to an appropriate location (likely using the file's name) after the file has finished transferring.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
gpGetFilePath	gpGetFilePathA	gpGetFilePathW

gpGetFilePathW and **gpGetFilePathA** are UNICODE and ANSI mapped versions of **gpGetFilePath**. The arguments of **gpGetFilePathA** are ANSI strings; those of **gpGetFilePathW** are wide-character strings.

Section Reference: [Gamespy Presence SDK](#)

gpGetFileProgress

This function is used to get the progress of a file being transferred.

```
GPResult gpGetFileProgress(  
    GPConnection * connection,  
    GPTransfer transfer,  
    int index,  
    int * progress );
```

Routine	Required Header	Distribution
gpGetFileProgress	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] A GP connection interface.

transfer

[in] A pointer to a GPTransfer object.

index

[in] The index of the file within the GPTransfer object.

progress

[in] The transfer progress.

Remarks

This function is used to get the progress of a file being transferred, or in other words, the number of bytes of the file either sent or received so far. If the file hasn't started transferring yet, the progress will be 0. The progress will be continually updated while the file is being transferred. If the file finishes transferring successfully, the progress should be the same as the file's size.

Section Reference: [Gamespy Presence SDK](#)

gpGetFileSize

This function is used to get the size of a file being transferred.

```
GPRResult gpGetFileSize(  
    GPCConnection * connection,  
    GPTransfer transfer,  
    int index,  
    int * size );
```

Routine	Required Header	Distribution
gpGetFileSize	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] A GP connection interface.

transfer

[in] A pointer to a GPTransfer object.

index

[in] The index of the file within the GPTransfer object.

size

[in] The size of the file.

Remarks

This function is used to get the size of a file being transferred. The size of each file is checked when the transfer is initialized, and this is the size that will be reported before the file is actually transferred. The size of the file is checked again when the file actually begins transferring, and this is the size that will be reported from that moment on (the two sizes will only be different if the file has changed during that time).

Section Reference: [Gamespy Presence SDK](#)

gpGetInfo

This function gets info on a particular profile.

```
GPRresult gpGetInfo(  
    GPConnection * connection,  
    GPProfile profile,  
    GPEnum checkCache,  
    GPEnum blocking,  
    GPCallback callback,  
    void * param );
```

Routine	Required Header	Distribution
gpGetInfo	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] A GP connection interface.

profile

[in] The profile ID of the user to get info on.

checkCache

[in] When set to GP_CHECK_CACHE the SDK will use the currently known info.

blocking

[in] GP_BLOCKING or GP_NON_BLOCKING

callback

[in] A user-supplied callback with an argument type of GPGetInfoResponseArg

param

[in] Pointer to user-defined data. This value will be passed unmodified to the callback function.

Remarks

This function gets profile info for the profile object profile. When the info has been retrieved, the callback will be called.. If info-caching is enabled, the info may be available locally, in which case it will be returned immediately if checkCache is GP_CHECK_CACHE. Otherwise, the server will be contacted for the info. If the server needs to be contacted, then the function will return immediately in non-blocking mode. If info-caching is enabled, any info retrieved from the server will be cached.

Section Reference: [Gamespy Presence SDK](#)

See Also: [GPGetInfoResponseArg](#)

gpGetLoginTicket

Retrieves a connection "token" that may be used by HTTP requests to uniquely identify the player.

```
GPRResult gpGetLoginTicket(  
    GPCConnection * connection,  
    char loginTicket[GP_LOGIN_TICKET_LEN] );
```

Routine	Required Header	Distribution
gpGetLoginTicket	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] A GP connection interface.

loginTicket

[out] The login ticket.

Remarks

Retrieves a connection "token" that may be used by HTTP requests to uniquely identify the player.

Section Reference: [Gamespy Presence SDK](#)

gpGetNumBlocked

Gets the total number of blocked players in the local profile's blocked list.

```
GPRResult gpGetNumBlocked(  
    GPCConnection * connection,  
    int * numBlocked );
```

Routine	Required Header	Distribution
gpGetNumBlocked	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] A GP connection interface.

numBlocked

[out] The total number of blocked players in the local profile's blocked list.

Remarks

This function will return 0 when GP is not connected. The blocked list is fully obtained after the login process is complete.

Section Reference: [Gamespy Presence SDK](#)

See Also: [gpGetBlockedProfile](#), [gplsBlocked](#)

gpGetNumBuddies

This function gets the number of buddies on the local profile's buddy list.

```
GPRResult gpGetNumBuddies(  
    GPConnection * connection,  
    int * numBuddies );
```

Routine	Required Header	Distribution
gpGetNumBuddies	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] A GP connection interface.

numBuddies

[out] The number of buddies.

Remarks

This function gets the number of buddies on the local profile's buddy list. It may take some time to receive the total number of buddies from the server, so this function may report a number smaller than the actual total while the complete buddy list is being received. To see the status of each buddy, call `gpGetBuddyStatus`.

The number of buddies is only valid until a buddy is added to or deleted from the buddy list.

Section Reference: [Gamespy Presence SDK](#)

gpGetNumFiles

This function is used to get the number of files (including directories) being transferred.

```
GPRResult gpGetNumFiles(  
    GPConnection * connection,  
    GPTransfer transfer,  
    int * num );
```

Routine	Required Header	Distribution
gpGetNumFiles	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] A GP connection interface.

transfer

[in] A GPTransfer object.

num

[out] The number of files within the GPTransfer object.

Remarks

This function is used to get the number of files being transferred. This total includes any directory names that are being sent.

Section Reference: [Gamespy Presence SDK](#)

gpGetNumTransfers

Returns the number of pending file transfers.

```
GPRResult gpGetNumTransfers(  
    GPConnection * connection,  
    int * num );
```

Routine	Required Header	Distribution
gpGetNumTransfers	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] A GP connection interface.

num

[out] The number of pending transfers.

Remarks

Returns the number of pending file transfers.

Section Reference: [Gamespy Presence SDK](#)

gpGetReverseBuddies

Get profiles that have you on their buddy list.

```
GPRResult gpGetReverseBuddies(  
    GPConnection * connection,  
    GPEnum blocking,  
    GPCallback callback,  
    void * param );
```

Routine	Required Header	Distribution
gpGetReverseBuddies	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] A GP connection interface.

blocking

[in] GP_BLOCKING or GP_NON_BLOCKING

callback

[in] A GP callback that will be passed a
GPGetReverseBuddiesResponseArg.

param

[in] Pointer to user-defined data. This value will be passed
unmodified to the callback function.

Remarks

Get profiles that have you on their buddy list.

Section Reference: [Gamespy Presence SDK](#)

See Also: [GPGetReverseBuddiesResponseArg](#)

gpGetTransfer

Returns the GPTransfer object at the specified index.

```
GPResult gpGetTransfer(  
    GPConnection * connection,  
    int index,  
    GPTransfer * transfer );
```

Routine	Required Header	Distribution
gpGetTransfer	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] A GP connection interface.

index

[in] Index of the GPTransfer object.

transfer

[out] A pointer to a GPTransfer object.

Remarks

Returns the GPTransfer object at the specified index.

Section Reference: [Gamespy Presence SDK](#)

gpGetTransferData

This function is used to retrieve arbitrary user-data stored with a transfer.

```
void * gpGetTransferData(  
    GPCConnection * connection,  
    GPTransfer transfer );
```

Routine	Required Header	Distribution
gpGetTransferData	<gp.h>	SDKZIP

Return Value

Parameters

connection

[in] A GP connection interface.

transfer

[in] A pointer to a GPTransfer object.

Remarks

This function allows an application to retrieve arbitrary user-data stored with a transfer.

Section Reference: [Gamespy Presence SDK](#)

gpGetTransferProfile

This function is used to get the remote profile for a transfer.

```
GPRResult gpGetTransferProfile(  
    GPCConnection * connection,  
    GPTransfer transfer,  
    GPProfile * profile );
```

Routine	Required Header	Distribution
gpGetTransferProfile	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] A GP connection interface.

transfer

[in] A pointer to a GPTransfer object.

profile

[out] The remote profile is stored here.

Remarks

This function is used to get the remote profile for a transfer.

Section Reference: [Gamespy Presence SDK](#)

gpGetTransferProgress

This function is used to get the total progress of the transfer, in bytes.

```
GPResult gpGetTransferProgress(  
    GPConnection * connection,  
    GPTransfer transfer,  
    int * progress );
```

Routine	Required Header	Distribution
gpGetTransferProgress	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] A GP connection interface.

transfer

[in] A pointer to a GPTransfer object.

progress

[out] The progress of the transfer, in bytes, is stored here.

Remarks

This function is used to determine the total progress of a file transfer. This is the total number of bytes of file data that have been transferred so far.

Section Reference: [Gamespy Presence SDK](#)

gpGetTransferSide

This function is used to get which side of the transfer the local profile is on (sending or receiving).

```
GPRResult gpGetTransferSide(  
    GPConnection * connection,  
    GPTransfer transfer,  
    GPEnum * side );
```

Routine	Required Header	Distribution
gpGetTransferSide	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] A GP connection interface.

transfer

[in] A pointer to a GPTransfer object.

side

[out] The side is stored here. This will be either
GP_TRANSFER_SENDER or GP_TRANSFER_RECEIVER

Remarks

This function is used to determine if the local profile is the sender or receiver for this transfer. This is often useful inside of the `gpTransferCallback` when dealing with a message that both the sender and receiver may get, such as `GP_FILE_END`.

Section Reference: [Gamespy Presence SDK](#)

gpGetTransferSize

This function is used to get the total size of the transfer, in bytes.

```
GPRResult gpGetTransferSize(  
    GPConnection * connection,  
    GPTransfer transfer,  
    int * size );
```

Routine	Required Header	Distribution
gpGetTransferSize	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] A GP connection interface.

transfer

[in] A pointer to a GPTransfer object.

size

[out] The size of the transfer, in bytes, will be stored here.

Remarks

This function is used to determine the total size of a file transfer. This is the sum of the sizes of all the files being transferred. When a file is transferred, its size may be different than the size originally reported for the file. This can cause the total size of the transfer to change during the course of the transfer.

Section Reference: [Gamespy Presence SDK](#)

gpGetTransferThrottle

This function can be used to get a transfer's throttle setting.

NOTE: Throttling is not currently implemented. Throttle information is transmitted between the local profile and remote profile, but no throttling actually occurs.

```
GPRResult gpGetTransferThrottle(  
    GPCConnection * connection,  
    GPTransfer transfer,  
    int * throttle );
```

Routine	Required Header	Distribution
gpGetTransferThrottle	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] A GP connection interface.

transfer

[in] A pointer to a GPTransfer object.

throttle

[out] The throttle setting is stored here.

Remarks

NOTE: Throttling is not currently implemented.

This function is used to get the throttle setting for a transfer. If throttle is positive, it is the throttle setting in bytes-per-second. If zero, the transfer is paused. If -1 , then there is no throttling.

Section Reference: [Gamespy Presence SDK](#)

gpGetUserNicks

This function gets the nicknames for a given e-mail/password (which identifies a user).

```
GPRresult gpGetUserNicks(  
    GPConnection * connection,  
    const gsi_char email[GP_EMAIL_LEN],  
    const gsi_char password[GP_PASSWORD_LEN],  
    GPEnum blocking,  
    GPCallback callback,  
    void * param );
```

Routine	Required Header	Distribution
gpGetUserNicks	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] A GP connection interface. (Does not have to be connected)

email

[in] The account e-mail address.

password

[in] The account password.

blocking

[in] GP_BLOCKING or GP_NON_BLOCKING

callback

[in] A user-supplied callback with an arg type of
GPGetUserNicksResponseArg

param

[in] Pointer to user-defined data. This value will be passed
unmodified to the callback function.

Remarks

This function contacts the Search Manager and gets a list of this user's nicks (profiles).

If you are unsure if the email address provided to this function is a valid email address, call `gplsValidEmail` first.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
gpGetUserNicks	gpGetUserNicksA	gpGetUserNicksW

gpGetUserNicksW and **gpGetUserNicksA** are UNICODE and ANSI mapped versions of **gpGetUserNicks**. The arguments of **gpGetUserNicksA** are ANSI strings; those of **gpGetUserNicksW** are wide-character strings.

Section Reference: [Gamespy Presence SDK](#)

See Also: [GPGetUserNicksResponseArg](#), [gplsValidEmail](#)

gpIDFromProfile

A GPProfile is now the same as a profileid.

```
GPRResult gpIDFromProfile(  
    GPConnection * connection,  
    GPProfile profile,  
    int * id );
```

Routine	Required Header	Distribution
gpIDFromProfile	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] A GP connection interface.

profile

[in] The GPProfile

id

[out] The profile ID of the GPProfile.

Remarks

A GPProfile is now the same as a profileid.

Section Reference: [Gamespy Presence SDK](#)

gpInitialize

This function is used to initialize a connection object.

```
GPRresult gpInitialize(  
    GPCConnection * connection,  
    int productID,  
    int namespaceID,  
    int partnerID );
```

Routine	Required Header	Distribution
gpInitialize	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] A GP connection interface.

productID

[in] The application's product ID. Contact devsupport@gamespy.com to obtain one.

namespaceID

[in] The application's namespace ID. 0 for no namespace, 1 for the default namespace, other numbers for custom namespaces.

partnerID

[in] The application's partner ID. Typically this will be set to the value defined by `GP_PARTNERID_GAMESPY`.

Remarks

This function initialize a connection object. As long as there are no errors, this object should stay valid until gpDestroy is called. After the object is initialized by this function, callbacks can be set for the connection, as well as any other states, such as info-caching.

namespaceID is normally 0 for no namespace or 1 for the default GameSpy namespace (used by GameSpy Arcade). If your game is using a custom namespace you can contact devsupport@gamespy.com to find out what namespace ID to use.

partnerID is normally the value defined by GP_PARTNERID_GAMESPY.

Section Reference: [Gamespy Presence SDK](#)

gpInvitePlayer

This function invites a player to play a certain game.

```
GPRResult gpInvitePlayer(  
    GPCConnection * connection,  
    GPProfile profile,  
    int productID,  
    const gsi_char location[GP_LOCATION_STRING_LEN] );
```

Routine	Required Header	Distribution
gpInvitePlayer	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] A GP connection interface.

profile

[in] The profile ID of the player to invite.

productID

[in] The product ID of the game to invite the player to.

location

[in] A message to send along with the invite. See remarks.

Remarks

This function is used to invite another profile to join the local profile in a game's title room. The remote profile will receive a GP_RECV_GAME_INVITE callback.

gpInvitePlayer may now take an optional text message to be sent along with the invite. This usually contains the server IP and other connecting information. This parameter may be NULL. The max length for the location info is 255 characters. When compiling in Unicode mode, the location will be converted to ASCII.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
gpInvitePlayer	gpInvitePlayerA	gpInvitePlayerW

gpInvitePlayerW and **gpInvitePlayerA** are UNICODE and ANSI mapped versions of **gpInvitePlayer**. The arguments of **gpInvitePlayerA** are ANSI strings; those of **gpInvitePlayerW** are wide-character strings.

Section Reference: [Gamespy Presence SDK](#)

gplsBlocked

Returns `gsi_true` if the given ProfileID is blocked, `gsi_false` if not blocked.

```
gsi_bool gplsBlocked(  
    GPConnection * connection,  
    GPProfile profile );
```

Routine	Required Header	Distribution
<code>gplsBlocked</code>	<code><gp.h></code>	SDKZIP

Return Value

Returns `gsi_true` if the given ProfileID is blocked, `gsi_false` if not blocked.

Parameters

connection

[in] A GP connection interface.

profile

[in] The profile ID of the player to check.

Section Reference: [Gamespy Presence SDK](#)

gplsBuddy

Returns 1 if the given ProfileID is a buddy, 0 if not a buddy.

```
int gplsBuddy(  
    GPConnection * connection,  
    GPProfile profile );
```

Routine	Required Header	Distribution
gplsBuddy	<gp.h>	SDKZIP

Return Value

Returns 1 if the given ProfileID is a buddy, 0 if not a buddy

Parameters

connection

[in] A GP connection interface.

profile

[in] The profile ID of the player to check.

Remarks

Returns 1 if the given ProfileID is a buddy, 0 if not a buddy.

Section Reference: [Gamespy Presence SDK](#)

gplsConnected

Determine whether the GPConnection object has established a connection with the server.

```
GPRresult gplsConnected(  
    GPConnection * connection,  
    GPEnum * connected );
```

Routine	Required Header	Distribution
gplsConnected	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] A GP connection interface.

connected

[out] The connected state. GP_CONNECTED or GP_NOT_CONNECTED. (See remarks.)

Remarks

If the connection parameter has not been initialized with `gpInitialize`, the connected parameter will be invalid and the return value will be `GP_PARAMETER_ERROR`.

Section Reference: [Gamespy Presence SDK](#)

gplsValidEmail

This function checks if there is an account with the given e-mail address.

```
GPResult gplsValidEmail(  
    GPConnection * connection,  
    const gsi_char email[GP_EMAIL_LEN],  
    GPEnum blocking,  
    GPCallback callback,  
    void * param );
```

Routine	Required Header	Distribution
gplsValidEmail	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] A GP connection interface.

email

[in] The e-mail address to list accounts for.

blocking

[in] GP_BLOCKING or GP_NON_BLOCKING

callback

[in] A user supplied callback with an arg of the type
GPIsValidEmailResponseArg

param

[in] Pointer to user defined data. This value will be passed
unmodified to the callback function.

Remarks

This function contacts the Search Manager and checks to see if there is a user with the given e-mail address.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
<code>gplsValidEmail</code>	<code>gplsValidEmailA</code>	<code>gplsValidEmailW</code>

gplsValidEmailW and **gplsValidEmailA** are UNICODE and ANSI mapped versions of **gplsValidEmail**. The arguments of **gplsValidEmailA** are ANSI strings; those of **gplsValidEmailW** are wide-character strings.

Section Reference: [Gamespy Presence SDK](#)

See Also: [GPIsValidEmailResponseArg](#)

gpNewProfile

This function creates a new profile for the local user.

```
GPResult gpNewProfile(  
    GPConnection * connection,  
    const gsi_char nick[GP_NICK_LEN],  
    GPEnum replace,  
    GPEnum blocking,  
    GPCallback callback,  
    void * param );
```

Routine	Required Header	Distribution
gpNewProfile	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] A GP connection interface.

nick

[in] The new profile nickname.

replace

[in] Replacement option. (See remarks.)

blocking

[in] GP_BLOCKING or GP_NON_BLOCKING

callback

[in] A user-supplied callback with an arg type of GPNewProfileResponseArg.

param

[in] Pointer to user defined data. This value will be passed unmodified to the callback function.

Remarks

This function creates a new profile for the local user. This function does not make the new profile the current profile. To switch to the newly created profile, the user must disconnect and then connect with the new nickname.

If the nick for the new profile is the same as the nick for an existing profile, an error will be generated, unless `replace` is `GP_REPLACE`. An application should use `GP_DONT_REPLACE` by default. If an error with the error code of `GP_NEWPROFILE_BAD_NICK` is received, this means that a profile with the provided nickname already exists. The application should at this point ask the user if he wants to replace the old profile. If the user does want to replace the old profile, **`gpNewProfile`** should be called again with `replace` set to `GP_REPLACE`.

When the new profile is created, the callback will be called.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
gpNewProfile	gpNewProfileA	gpNewProfileW

gpNewProfileW and **gpNewProfileA** are UNICODE and ANSI mapped versions of **gpNewProfile**. The arguments of **gpNewProfileA** are ANSI strings; those of **gpNewProfileW** are wide-character strings.

Section Reference: [Gamespy Presence SDK](#)

See Also: [GPNewProfileResponseArg](#)

gpNewUser

This function creates a new user account and a profile in that account. Unlike gpConnectNewUser, gpNewUser does not login with the new account. The local user does not need to be connected to use this function.

```
GPResult gpNewUser(  
    GPConnection * connection,  
    const gsi_char nick[GP_NICK_LEN],  
    const gsi_char uniquenick[GP_UNIQUENICK_LEN],  
    const gsi_char email[GP_EMAIL_LEN],  
    const gsi_char password[GP_PASSWORD_LEN],  
    const gsi_char cdkey[GP_CDKEY_LEN],  
    GPEnum blocking,  
    GPCallback callback,  
    void * param );
```

Routine	Required Header	Distribution
gpNewUser	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] The connection on which to create the new user.

nick

[in] The desired nickname for the initial profile of the new account.

uniquenick

[in] The desired unquenick for the initial profile of the new account.

email

[in] The desired e-mail for the initial profile of the new account.

password

[in] The desired password for the initial profile of the new account.

cdkey

[in] An optional CDKey to associate with the unquenick.

blocking

[in] GP_BLOCKING or GP_NON_BLOCKING

callback

[in] User supplied callback function with an arg type of GPNewUserResponseArg.

param

[in] Pointer to user defined data. This value will be passed unmodified to the callback function.

Remarks

This function creates a new user and profile. The nick, email, and password are required parameters, uniquenick and cdkey are optional. This function is similar to gpConnectNewUser, however it does not connect the new user. You do not need to be connected to use this function.

When the new user is created, the callback will be called.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
gpNewUser	gpNewUserA	gpNewUserW

gpNewUserW and **gpNewUserA** are UNICODE and ANSI mapped versions of **gpNewUser**. The arguments of **gpNewUserA** are ANSI strings; those of **gpNewUserW** are wide-character strings.

Section Reference: [Gamespy Presence SDK](#)

See Also: [GPNewUserResponseArg](#)

gpProcess

This function does any necessary processing that needs to be done on connection.

```
GPRresult gpProcess(  
    GPCConnection * connection );
```

Routine	Required Header	Distribution
gpProcess	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] A GP connection interface.

Remarks

This function does any necessary processing that needs to be done on connection. This includes checking for buddy messages, checking for buddy status changes, and completing any non-blocking operations. This functions should be called frequently, typically in the application's main loop. If an operation is finished during a call to this function, any necessary callbacks will be called.

Section Reference: [Gamespy Presence SDK](#)

gpProfileFromID

Translates a profile id into a GPProfile.

```
GPRResult gpProfileFromID(  
    GPConnection * connection,  
    GPProfile * profile,  
    int id );
```

Routine	Required Header	Distribution
gpProfileFromID	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] A GP connection interface.

profile

[out] The GPProfile for the given profile ID.

id

[in] The profile ID.

Remarks

This function is no longer needed. GPProfiles are now the same as profile id's.

Section Reference: [Gamespy Presence SDK](#)

gpProfileSearch

This function searches for profiles based on certain criteria.

```
GPResult gpProfileSearch(  
    GPConnection * connection,  
    const gsi_char nick[GP_NICK_LEN],  
    const gsi_char uniquenick[GP_UNIQUENICK_LEN],  
    const gsi_char email[GP_EMAIL_LEN],  
    const gsi_char firstname[GP_FIRSTNAME_LEN],  
    const gsi_char lastname[GP_LASTNAME_LEN],  
    int icquin,  
    GPEnum blocking,  
    GPCallback callback,  
    void * param );
```

Routine	Required Header	Distribution
gpProfileSearch	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] A GP connection interface.

nick

[in] If not NULL or "", search for profiles with this nick.

uniquenick

[in] If not NULL or "", search for profiles with this uniquenick.

email

[in] If not NULL or "", search for profiles with this email.

firstname

[in] If not NULL or "", search for profiles with this firstname.

lastname

[in] If not NULL or "", search for profiles with this lastname.

icquin

[in] If not 0, search for profiles with this icquin.

blocking

[in] GP_BLOCKING or GP_NON_BLOCKING

callback

[in] A user supplied callback with an arg type of GPProfileSearchResponseArg.

param

[in] Pointer to user defined data. This value will be passed unmodified to the callback function.

Remarks

This function contacts the Search Manager and attempts to find all profiles that match the search criteria. A profile matches the provided search criteria only if its corresponding values are the same as those provided. Currently, there is no substring matching, and the criteria is case-sensitive.

When the search is complete, the callback will be called.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
gpProfileSearch	gpProfileSearchA	gpProfileSearchW

gpProfileSearchW and **gpProfileSearchA** are UNICODE and ANSI mapped versions of **gpProfileSearch**. The arguments of **gpProfileSearchA** are ANSI strings; those of **gpProfileSearchW** are wide-character strings.

Section Reference: [Gamespy Presence SDK](#)

See Also: [GPProfileSearchResponseArg](#)

gpProfilesReport

Debug function to dump information on known profiles to the console.

```
void gpProfilesReport(  
    GPConnection * connection,  
    void (*)(const char * output) report );
```

Routine	Required Header	Distribution
gpProfilesReport	<gp.h>	SDKZIP

Parameters

connection

[in] A GP connection interface.

report

[in] A user-supplied function to be triggered with each line of info.
See remarks.

Remarks

This is a debug-only function that will dump the contents of the internal profile map to the user-supplied function.

The user-supplied function is most commonly printf.

Section Reference: [Gamespy Presence SDK](#)

gpRegisterCdKey

This function attempts to register a cdkey and associate it with the local profile.

```
GPRResult gpRegisterCdKey(  
    GPConnection * connection,  
    const gsi_char cdkey[GP_CDKEY_LEN],  
    GPEnum blocking,  
    GPCallback callback,  
    void * param );
```

Routine	Required Header	Distribution
gpRegisterCdKey	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] A GP connection interface.

cdkey

[in] A CDKey to associate with the local profile..

blocking

[in] GP_BLOCKING or GP_NON_BLOCKING

callback

[in] A user supplied callback with an arg type of GPRegisterCdKeyResponseArg.

param

[in] Pointer to user-defined data. This value will be passed unmodified to the callback function.

Remarks

This function will only work if GP is connected. Note that only one CDKey can be associated with a single profile. Once a CDKey has been associated, it cannot be associated with any other profiles.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
gpRegisterCdKey	gpRegisterCdKeyA	gpRegisterCdKeyW

gpRegisterCdKeyW and **gpRegisterCdKeyA** are UNICODE and ANSI mapped versions of **gpRegisterCdKey**. The arguments of **gpRegisterCdKeyA** are ANSI strings; those of **gpRegisterCdKeyW** are wide-character strings.

Section Reference: [Gamespy Presence SDK](#)

See Also: [GPRegisterCdKeyResponseArg](#)

gpRegisterUniqueNick

This function attempts to register a uniquenick and associate it with the local profile.

```
GPRResult gpRegisterUniqueNick(  
    GPConnection * connection,  
    const gsi_char uniquenick[GP_UNIQUENICK_LEN],  
    const gsi_char cdkey[GP_CDKEY_LEN],  
    GPEnum blocking,  
    GPCallback callback,  
    void * param );
```

Routine	Required Header	Distribution
gpRegisterUniqueNick	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] A GP connection interface.

uniquenick

[in] The desired uniquenick It can be up to GP_UNIQUENICK_LEN characters long, including the NUL.

cdkey

[in] An optional CDKey to associate with the uniquenick. If not using CDKeys this should be NULL.

blocking

[in] GP_BLOCKING or GP_NON_BLOCKING

callback

[in] A user supplied callback with an arg type of GPRegisterUniqueNickResponseArg.

param

[in] Pointer to user-defined data. This value will be passed unmodified to the callback function.

Remarks

This function attempts to register a uniquenick and associate it with the local profile. It should only be used if the namespaceID passed to gplInitialize was greater than 0. The backend makes certain checks on a uniquenick before it is allowed to be registered. For details on what is checked, see the "Uniquenick Checks" section at the bottom of "GameSpy Presence SDK.doc".

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
gpRegisterUniqueNick	gpRegisterUniqueNickA	gpRegisterUniqueNickW

gpRegisterUniqueNickW and **gpRegisterUniqueNickA** are UNICODE and ANSI mapped versions of **gpRegisterUniqueNick**. The arguments of **gpRegisterUniqueNickA** are ANSI strings; those of **gpRegisterUniqueNickW** are wide-character strings.

Section Reference: [Gamespy Presence SDK](#)

See Also: [GPRegisterUniqueNickResponseArg](#)

gpRejectTransfer

This function is used to reject a file transfer request.

```
GPResult gpRejectTransfer(  
    GPConnection * connection,  
    GPTransfer transfer,  
    const gsi_char * message );
```

Routine	Required Header	Distribution
gpRejectTransfer	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] A GP connection interface.

transfer

[in] A pointer to a GPTransfer object.

message

[in] An optional message to send along with the rejection.

Remarks

This function is used to reject an incoming files request. This will also free the transfer, so it should not be referenced again once rejected.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
gpRejectTransfer	gpRejectTransferA	gpRejectTransferW

gpRejectTransferW and **gpRejectTransferA** are UNICODE and ANSI mapped versions of **gpRejectTransfer**. The arguments of **gpRejectTransferA** are ANSI strings; those of **gpRejectTransferW** are wide-character strings.

Section Reference: [Gamespy Presence SDK](#)

See Also: [gpSendFiles](#), [gpAcceptTransfer](#)

gpRemoveFromBlockedList

Removes a remote profile from the local player's blocked list.

```
GPRResult gpRemoveFromBlockedList(  
    GPConnection * connection,  
    GPProfile profile );
```

Routine	Required Header	Distribution
gpRemoveFromBlockedList	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] A GP connection interface.

profile

[in] The profileid of the player to be removed from the blocked list.

Remarks

A blocked player is essentially invisible to player who has him/her blocked. The local player will not receive any communication from the blocked player, nor will the local player be able to contact the blocked player in any way.

This function will only work when GP is connected. This function will not return any callback upon success, but the GP_ERROR callback will be called should an error occur during the removal attempt.

Section Reference: [Gamespy Presence SDK](#)

See Also: [gpAddToBlockedList](#), [gpGetNumBlocked](#), [gpGetBlockedProfile](#), [gplsBlocked](#)

gpRevokeBuddyAuthorization

Remove the local client from a remote users buddy list.

```
GPRresult gpRevokeBuddyAuthorization(  
    GPCConnection * connection,  
    GPProfile profile );
```

Routine	Required Header	Distribution
gpRevokeBuddyAuthorization	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] A GP connection interface.

profile

[in] The profile ID of the remote player.

Remarks

Use this function when the local client no longer wants the remote player to be able to send buddy messages or view status info.

Section Reference: [Gamespy Presence SDK](#)

gpSendBuddyMessage

This function sends a message to a buddy.

```
GPRResult gpSendBuddyMessage(  
    GPConnection * connection,  
    GPProfile profile,  
    const gsi_char * message );
```

Routine	Required Header	Distribution
gpSendBuddyMessage	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] A GP connection interface.

profile

[in] The profile object for the buddy to whom the message is going.

message

[in] A user-readable text string containing the message to send to the buddy.

Remarks

If the buddy is not behind a firewall, and a direct connection is possible, the message can be any size. However, if the message needs to be sent through the server (i.e., the buddy is behind a firewall), then the message needs to be sent through the server. In this case, there is a limit of 1024 characters. Any message longer than 1024 characters, that needs to be sent through the server, will be truncated without warning or notice.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
gpSendBuddyMessage	gpSendBuddyMessageA	gpSendBuddyMessageW

gpSendBuddyMessageW and **gpSendBuddyMessageA** are UNICODE and ANSI mapped versions of **gpSendBuddyMessage**. The arguments of **gpSendBuddyMessageA** are ANSI strings; those of **gpSendBuddyMessageW** are wide-character strings.

Section Reference: [Gamespy Presence SDK](#)

gpSendBuddyRequest

This function sends a request to a remote profile to ask for permission to add the remote profile to the local profile's buddy list.

```
GPRResult gpSendBuddyRequest(  
    GPConnection * connection,  
    GPProfile profile,  
    const gsi_char reason[GP_REASON_LEN] );
```

Routine	Required Header	Distribution
gpSendBuddyRequest	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] A GP connection interface.

profile

[in] The remote profile to which the buddy request is being made.

reason

[in] A text string that (optionally) explains why the user is making the buddy request.

Remarks

This function sends a request to the given remote profile, asking if the local profile can make the remote profile a buddy. There is no immediate response to this message. If the remote profile authorizes the request, a buddy message and a status message will be received from the new buddy. However, this can take any amount of time.

This message causes the `gpRecvBuddyRequest` callback to be called for the remote profile.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
gpSendBuddyRequest	gpSendBuddyRequestA	gpSendBuddyRequestW

gpSendBuddyRequestW and **gpSendBuddyRequestA** are UNICODE and ANSI mapped versions of **gpSendBuddyRequest**. The arguments of **gpSendBuddyRequestA** are ANSI strings; those of **gpSendBuddyRequestW** are wide-character strings.

Section Reference: [Gamespy Presence SDK](#)

gpSendBuddyUTM

Sends a UTM (under-the-table) message to a buddy.

```
GPRresult gpSendBuddyUTM(  
    GPConnection * connection,  
    GPProfile profile,  
    const gsi_char * message,  
    int sendOption );
```

Routine	Required Header	Distribution
gpSendBuddyUTM	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] A GP connection interface.

profile

[in] The profile object for the buddy to whom the message is going.

message

[in] A user-readable text string containing the message to send to the buddy.

sendOption

[in] UTM sending options - defined in GPEnum. Pass in 0 for no options.

Remarks

If the buddy is not behind a firewall, and a direct connection is possible, the message can be any size. However, if the message needs to be sent through the server (i.e., the buddy is behind a firewall), then the message needs to be sent through the server. In this case, there is a limit of 1024 characters. Any message longer than 1024 characters, that needs to be sent through the server, will be truncated without warning or notice.

If GP_DONT_ROUTE is listed as a sendOption, the SDK will only attempt to send this message directly to the player and not route it through the server.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
gpSendBuddyUTM	gpSendBuddyUTMA	gpSendBuddyUTMW

gpSendBuddyUTMW and **gpSendBuddyUTMA** are UNICODE and ANSI mapped versions of **gpSendBuddyUTM**. The arguments of **gpSendBuddyUTMA** are ANSI strings; those of **gpSendBuddyUTMW** are wide-character strings.

Section Reference: [Gamespy Presence SDK](#)

gpSendFiles

This function attempts to send one or more files (and/or sub-directory names) to another profile.

```
GPRResult gpSendFiles(  
    GPConnection * connection,  
    GPTransfer * transfer,  
    GPProfile profile,  
    const gsi_char * message,  
    gpSendFilesCallback callback,  
    void * param );
```

Routine	Required Header	Distribution
gpSendFiles	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] A GP connection interface.

transfer

[out] A pointer to a GPTransfer object.

profile

[in] The profile to send to. Must be a buddy, or we must be his buddy.

message

[in] An optional message to send along with the request.

callback

[in] This callback will get called repeatedly to get the list of files to send. See below.

param

[in] Pointer to user-defined data. This value will be passed unmodified to the callback function.

Remarks

This function attempts to send files to a remote profile. The profile must be either on the local profile's buddy list, or the local profile must be on the remote profile's buddy list. To send the files, a direct connection must be established between the two profiles. If both are behind firewalls, or a direct connection cannot be established for any other reason, the transfer will fail.

A successful call to this function will create a transfer object (which is identified by `transfer`). This object will not be freed until either the connection is destroyed with `gpDestroy()`, or the object is explicitly freed with `gpFreeTransfer()`. The object is not automatically freed when the transfer completes. Information about this transfer will be passed back to the application through the `GP_TRANSFER_CALLBACK` callback.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
gpSendFiles	gpSendFilesA	gpSendFilesW

gpSendFilesW and **gpSendFilesA** are UNICODE and ANSI mapped versions of **gpSendFiles**. The arguments of **gpSendFilesA** are ANSI strings; those of **gpSendFilesW** are wide-character strings.

Section Reference: [Gamespy Presence SDK](#)

gpSetCallback

This function is used to set callbacks. The callbacks that get set with this function are called as a result of data received from the server, such as messages or status updates.

```
GPRResult gpSetCallback(  
    GPCConnection * connection,  
    GPEnum func,  
    GPCallback callback,  
    void * param );
```

Routine	Required Header	Distribution
gpSetCallback	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] A GP connection interface.

func

[in] An enum that indicates which callback is being set.

callback

[in] The user-supplied function that will be called.

param

[in] Pointer to user-defined data. This value will be passed unmodified to the callback function.

Remarks

This function sets what callback to call when data is received from the server, such as messages or status updates, or an error is generated. If no callback is set for a certain situation, then no alert will be given when that situation occurs. For example, if no GP_RECV_BUDDY_REQUEST callback is set, then there will be no way of detecting when a remote profile wants to add the local profile as a buddy.

These callbacks can be generated during any function that checks for data received from the server, typically gpProcess or a blocking operation function.

The following can be used as parameters for callback type:

GP_ERROR,
GP_RECV_BUDDY_REQUEST,
GP_RECV_BUDDY_STATUS,
GP_RECV_BUDDY_MESSAGE,
GP_RECV_GAME_INVITE,
GP_TRANSFER_CALLBACK,
GP_RECV_BUDDY_AUTH,
GP_RECV_BUDDY_REVOKE.

Section Reference: [Gamespy Presence SDK](#)

gpSetInfoCacheFilename

Sets the file name for the internal profile cache.

```
void gpSetInfoCacheFilename(  
    const gsi_char * filename );
```

Routine	Required Header	Distribution
gpSetInfoCacheFilename	<gp.h>	SDKZIP

Parameters

filename

[in] The filename to use for the profile cache.

Remarks

This function should be called before `gplInitialize`.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Def
gpSetInfoCacheFilename	gpSetInfoCacheFilenameA	gpSetInfoCacheFile

gpSetInfoCacheFilenameW and **gpSetInfoCacheFilenameA** are UNICODE and ANSI mapped versions of **gpSetInfoCacheFilename**. The arguments of **gpSetInfoCacheFilenameA** are ANSI strings; those of **gpSetInfoCacheFilenameW** are wide-character strings.

Section Reference: [Gamespy Presence SDK](#)

gpSetInfod

These functions are used to set local info.

```
GPRresult gpSetInfod(  
    GPConnection * connection,  
    GPEnum info,  
    int day,  
    int month,  
    int year );
```

Routine	Required Header	Distribution
gpSetInfod	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] A GP connection interface.

info

[in] An enum indicating what info to update.

day

[in] The day.

month

[in] The month.

year

[in] The year.

Remarks

These functions are used to set local info. The info does not actually get updated (sent to the server) until the next call to gpProcess. If a string is longer than the allowable length for that info, it will be truncated without warning.

Section Reference: [Gamespy Presence SDK](#)

gpSetInfo

These functions are used to set local info.

```
GPRresult gpSetInfo(  
    GPConnection * connection,  
    GPEnum info,  
    int value );
```

Routine	Required Header	Distribution
gpSetInfo	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] A GP connection interface.

info

[in] An enum indicating what info to update.

value

[in] The integer value.

Remarks

These functions are used to set local info. The info does not actually get updated (sent to the server) until the next call to gpProcess. If a string is longer than the allowable length for that info, it will be truncated without warning.

Section Reference: [Gamespy Presence SDK](#)

gpSetInfoMask

```
GPRResult gpSetInfoMask(  
    GPConnection * connection,  
    GPEnum mask );
```

Routine	Required Header	Distribution
gpSetInfoMask	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] A GP connection interface.

mask

[in] The info type. See remarks.

Remarks

The possible mask values are:

GP_MASK_NONE
GP_MASK_HOMEPAGE
GP_MASK_ZIPCODE
GP_MASK_COUNTRYCODE
GP_MASK_BIRTHDAY
GP_MASK_SEX
GP_MASK_EMAIL
GP_MASK_ALL.

Section Reference: [Gamespy Presence SDK](#)

gpSetInfos

These functions are used to set local info.

```
GPRresult gpSetInfos(  
    GPConnection * connection,  
    GPEnum info,  
    const gsi_char * value );
```

Routine	Required Header	Distribution
gpSetInfos	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] A GP connection interface.

info

[in] An enum indicating what info to update.

value

[in] The string value.

Remarks

These functions are used to set local info. The info does not actually get updated (sent to the server) until the next call to gpProcess.

If a string is longer than the allowable length for that info, it will be truncated without warning.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
gpSetInfos	gpSetInfosA	gpSetInfosW

gpSetInfosW and **gpSetInfosA** are UNICODE and ANSI mapped versions of **gpSetInfos**. The arguments of **gpSetInfosA** are ANSI strings; those of **gpSetInfosW** are wide-character strings.

Section Reference: [Gamespy Presence SDK](#)

gpSetStatus

This function sets the local profile's status.

```
GPRResult gpSetStatus(  
    GPConnection * connection,  
    GPEnum status,  
    const gsi_char statusString[GP_STATUS_STRING_LEN],  
    const gsi_char locationString[GP_LOCATION_STRING_LEN] );
```

Routine	Required Header	Distribution
gpSetStatus	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] A GP connection interface.

status

[in] An enum indicating the status to set.

statusString

[in] A text string with a user-readable explanation of the status.

locationString

[in] A URL indicating the local profile's location, in the form "gamename://IP.address:port/extra/info".

Remarks

This function sets the local profile's status. The status consists of an enum specifying a mode (online, offline, playing, etc.), a text explanation of the status, and a URL specifying a location and protocol (e.g., "quake://12.34.56.78:9999").

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
gpSetStatus	gpSetStatusA	gpSetStatusW

gpSetStatusW and **gpSetStatusA** are UNICODE and ANSI mapped versions of **gpSetStatus**. The arguments of **gpSetStatusA** are ANSI strings; those of **gpSetStatusW** are wide-character strings.

Section Reference: [Gamespy Presence SDK](#)

gpSetTransferData

This function is used to store arbitrary user-data with a transfer.

```
GPRResult gpSetTransferData(  
    GPConnection * connection,  
    GPTransfer transfer,  
    void * userData );
```

Routine	Required Header	Distribution
gpSetTransferData	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] A GP connection interface.

transfer

[in] A pointer to a GPTransfer object.

userData

[in] Arbitrary user data to associate with the transfer.

Remarks

This function allows an application to associate arbitrary user-data with a transfer.

Section Reference: [Gamespy Presence SDK](#)

gpSetTransferDirectory

This function can be used to set the directory that files are received into.

```
GPRresult gpSetTransferDirectory(  
    GPConnection * connection,  
    GPTransfer transfer,  
    const gsi_char * directory );
```

Routine	Required Header	Distribution
gpSetTransferDirectory	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] A GP connection interface.

transfer

[in] A pointer to a GPTransfer object.

directory

[in] The directory to store received files in. This must be the path to a directory, and it must end in a slash or backslash.

Remarks

This allows the application to set which directory received files are stored in. They will all be stored in this directory, with names randomly generated by GP. It is then up to the application to place the files in the appropriate directories with the appropriate names.

If no directory is set, a directory will be picked. On win32, the GetTempPath function is used to pick a directory. If the application wants to set a directory explicitly, it must call this function before accepting the transfer. The function will fail if it is called after the transfer has started. This function only sets the directory for the specified transfer, and for no others.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
gpSetTransferDirectory	gpSetTransferDirectoryA	gpSetTransferDirectoryW

gpSetTransferDirectoryW and **gpSetTransferDirectoryA** are UNICODE and ANSI mapped versions of **gpSetTransferDirectory**. The arguments of **gpSetTransferDirectoryA** are ANSI strings; those of **gpSetTransferDirectoryW** are wide-character strings.

Section Reference: [Gamespy Presence SDK](#)

gpSetTransferThrottle

This function can be used to set a throttle on a transfer.

NOTE: Throttling is not currently implemented. Throttle information is transmitted between the local profile and remote profile, but no throttling actually occurs.

```
GPRResult gpSetTransferThrottle(  
    GPConnection * connection,  
    GPTransfer transfer,  
    int throttle );
```

Routine	Required Header	Distribution
gpSetTransferThrottle	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] A GP connection interface.

transfer

[in] A pointer to a GPTransfer object.

throttle

[in] The throttle setting.

Remarks

NOTE: Throttling is not currently implemented.

This function can be used to either pause a transfer or limit a transfer to a maximum rate. It can be called by either the sender or the receiver. After a call to this function, both the local profile and remote profile will receive a gpTransferCallback of type GP_TRANSFER_THROTTLE.

Section Reference: [Gamespy Presence SDK](#)

gpSkipFile

This function is used to skip transferring a certain file.

```
GPRResult gpSkipFile(  
    GPConnection * connection,  
    GPTransfer transfer,  
    int index );
```

Routine	Required Header	Distribution
gpSkipFile	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] A GP connection interface.

transfer

[in] A pointer to a GPTransfer object.

index

[in] Index of the file within the GPTransfer object.

Remarks

This function is used to skip a file in the transfer. It can be called either before a file is transferred, or while a file is being transferred. If it is called before the file starts transferring, then the a GP_FILE_SKIP callback will be received when the file becomes the current file. If it is called while a file is being transferred, then the GP_FILE_SKIP will be called as soon as possible, and the file will stop transferring.

Section Reference: [Gamespy Presence SDK](#)

gpSuggestUniqueNick

This function gets suggested uniquenicks from the backend.

```
GPRresult gpSuggestUniqueNick(  
    GPConnection * connection,  
    const gsi_char desirednick[GP_UNIQUNICK_LEN],  
    GPEnum blocking,  
    GPcallback callback,  
    void * param );
```

Routine	Required Header	Distribution
gpSuggestUniqueNick	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] A GP connection interface.

desirednick

[in] The desired uniquenick It can be up to GP_UNIQUENICK_LEN characters long, including the NUL.

blocking

[in] GP_BLOCKING or GP_NON_BLOCKING

callback

[in] A user supplied callback with an arg type of GPSuggestUniqueNickResponseArg.

param

[in] Pointer to user defined data. This value will be passed unmodified to the callback function.

Remarks

This function gets a set of suggested nicks based on the desirednick. A request is sent to the backend for suggestions based on the provided desirednick. After getting a response, the callback is called with a list of uniquenicks based on the desirednick. These suggested uniquenicks can then be used in a call to gpNewUser, gpRegisterUniqueNick, or gpSetInfos.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
gpSuggestUniqueNick	gpSuggestUniqueNickA	gpSuggestUniqueNickW

gpSuggestUniqueNickW and **gpSuggestUniqueNickA** are UNICODE and ANSI mapped versions of **gpSuggestUniqueNick**. The arguments of **gpSuggestUniqueNickA** are ANSI strings; those of **gpSuggestUniqueNickW** are wide-character strings.

Section Reference: [Gamespy Presence SDK](#)

See Also: [GPSuggestUniqueNickResponseArg](#)

gpUserIDFromProfile

This function gets a profile's user ID.

```
GPRResult gpUserIDFromProfile(  
    GPConnection * connection,  
    GPProfile profile,  
    int * userid );
```

Routine	Required Header	Distribution
gpUserIDFromProfile	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] A GP connection interface.

profile

[in] The profile ID.

userid

[out] The user ID associated with the specified profile ID.

Remarks

Every profile is associated with a user account, and each user account has a user id associated with it. This functions gets the user id for a given profile's user account.

Section Reference: [Gamespy Presence SDK](#)

Presence and Messaging SDK Callbacks

[GPCallback](#)

A generic callback definition used to specify the callback supplied to GP SDK functions.

[gpSendFilesCallback](#)

This is a callback used by gpSendFiles() to get the list of files to send.

GPCallback

A generic callback definition used to specify the callback supplied to GP SDK functions.

```
typedef void (*GPCallback)(  
    GPConnection * connection,  
    void * arg,  
    void * param );
```

Routine	Required Header	Distribution
GPCallback	<gp.h>	SDKZIP

Parameters

connection

[in] The connection associated with the task.

arg

[in] Pointer to a response structure whose content depends on the task in progress.

param

[in] User-data that was passed into the original function.

Remarks

The *arg* parameter content varies depending on the task. For example, a callback that is specified when calling **gpGetInfo()** should cast its incoming arg pointer to type **GPGetInfoResponseArg**.

Section Reference: [Gamespy Presence SDK](#)

See Also: [gpGetInfo](#), [GPGetInfoResponseArg](#)

gpSendFilesCallback

This is a callback used by gpSendFiles() to get the list of files to send.

```
typedef void (*gpSendFilesCallback)(  
    GPCConnection * connection,  
    int index,  
    const gsi_char ** path,  
    const gsi_char ** name,  
    void * param );
```

Routine	Required Header	Distribution
gpSendFilesCallback	<gp.h>	SDKZIP

Parameters

connection

[in] The connection the files are to be sent on.

index

[in] This starts at 0 and is incremented by 1 each time the callback gets called.

path

[in] Point this to the path to the file to send, or NULL for a directory.

name

[in] Point this to the name to send the file under, or NULL to use the file's local name.

param

[in] User-data that was passed into gpSendFiles.

Remarks

This function will be called repeatedly until neither path or name are set (or both set to NULL). The callback can be used to specify either a file or a directory. To specify a file, set path to point to a string containing the path to the file. If the name is also set, then it contains the name to send the file under. The name can either be a simple filename ("file.ext"), or it can contain a path ("files/file.ext"). If a path is specified, and name is not set (or set to NULL), then the filename part of the path will be used. For example, if path points to "c:\files\file.ext" and name is not set, then the name will be "file.ext".

To specify a directory, don't set the path (or set it to NULL), then set a name. The name will be treated as a directory to create. For example, if path is not set, and name is "files", this instructs the receiver to create a directory named "files".

The name for a file or folder cannot contain any directory level references (e.g., "../file.exe"), cannot start with a slash or backslash, cannot contain any empty directory names in the path, and cannot contain any invalid characters (: * ? " < > | \n).

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
gpSendFilesCallback	gpSendFilesCallbackA	gpSendFilesCallbackW

gpSendFilesCallbackW and **gpSendFilesCallbackA** are UNICODE and ANSI mapped versions of **gpSendFilesCallback**. The arguments of **gpSendFilesCallbackA** are ANSI strings; those of **gpSendFilesCallbackW** are wide-character strings.

Section Reference: [Gamespy Presence SDK](#)

Presence and Messaging SDK Structures

GPBuddyStatus	Buddy status.
GPCheckResponseArg	The arg parameter passed to a callback generated by a call to gpCheckUser is of this type.
GPConnectResponseArg	The arg parameter passed through to a GP_CALLBACK call after attempting to connect is of this type.
GPDeleteProfileResponseArg	This arg data type contains the data for a delete profile operation. It is generated by a call to the callback passed to gpDeleteProfile.
GPErrorArg	Contains information about an error which has occurred.
GPFindPlayerMatch	An element of GPFindPlayersResponseArg, which is the arg parameter passed to a callback generated by a call to gpFindPlayers .
GPFindPlayersResponseArg	The arg parameter passed to a callback generated by a call to gpFindPlayers is of this type.
GPGetInfoResponseArg	The arg parameter passed to a

	callback generated by a call to gpGetInfo is of this type. The structure provides information about the specified profile.
GPGetReverseBuddiesResponseArg	The arg parameter passed to a callback generated by a call to gpGetReverseBuddies is of this type
GPGetUserNicksResponseArg	The arg parameter passed to a callback generated by a call to gpGetUserNicks is of this type.
GPIsValidEmailResponseArg	The arg parameter passed to a callback generated by a call to gplsValidEmail is of this type.
GPNewProfileResponseArg	The arg parameter passed to a callback generated by a call to gpNewProfile is of this type.
GPNewUserResponseArg	The arg parameter passed to a callback generated by a call to gpNewUser is of this type.
GPProfileSearchMatch	Information about a profile which is returned by a requested search. Is often collected in a list, such as those found in GPGetReverseBuddiesResponseArg or GPProfileSearchResponseArg,
GPProfileSearchResponseArg	The arg parameter passed to a

	callback generated by a call to gpProfileSearch is of this type. Contains information about the profiles that matched the search criteria.
GPRecvBuddyAuthArg	Information sent to the GP_RECV_BUDDY_AUTH callback
GPRecvBuddyMessageArg	Information sent to the GP_RECV_BUDDY_MESSAGE callback.
GPRecvBuddyRequestArg	Information sent to the GP_RECV_BUDDY_REQUEST callback.
GPRecvBuddyRevokeArg	Information sent to the GP_RECV_BUDDY_REVOKE callback.
GPRecvBuddyStatusArg	Information sent to the GP_RECV_BUDDY_STATUS callback.
GPRecvGameInviteArg	Information sent to the GP_RECV_GAME_INVITE callback
GPRegisterCdKeyResponseArg	The arg parameter passed to a callback generated by a call to gpRegisterCdKey is of this type
GPRegisterUniqueNickResponseArg	

The arg parameter passed to a callback generated by a call to gpRegisterUniqueNick is of this type

[GPSuggestUniqueNickResponseArg](#)

The arg parameter passed to a callback generated by a call to gpSuggestUniqueNick is of this type

[GPTransferCallbackArg](#)

The arg parameter passed to a Transfer Callback .

GPBuddyStatus

Buddy status.

```
typedef struct  
{  
    GPProfile profile;  
    GPEnum status;  
    gsi_char statusString[GP_STATUS_STRING_LEN];  
    gsi_char locationString[GP_LOCATION_STRING_LEN];  
    unsigned int ip;  
    int port;  
} GPBuddyStatus;
```

Members

profile

The profile of the buddy.

status

A value of GPEnum which represents the "Status" of the buddy.

statusString

The buddy "Status" in user-readable form.

locationString

URL indicating the location of the buddy. It is of the form "gamename://IP.address:port/extra/info".

ip

The buddy's IP address in network byte order (big-endian). This is used for buddy-to-buddy messaging.

port

The buddy's TCP listening port. If this is 0, the buddy is behind a firewall. This is used for buddy-to-buddy messaging.

Remarks

Possible values for the "status" field are:

GP_ONLINE

GP_OFFLINE

GP_PLAYING

GP_STAGING

GP_CHATTING

See the Status section of GPEnum for details.

Section Reference: [Gamespy Presence SDK](#)

See Also: [GPEnum](#), [gpGetBuddyStatus](#)

GPCheckResponseArg

The arg parameter passed to a callback generated by a call to gpCheckUser is of this type.

```
typedef struct  
{  
    GPResult result;  
    GPProfile profile;  
} GPCheckResponseArg;
```

Members

result

Result of the check; GP_NO_ERROR if successful.

profile

Profile for the user being checked.

Section Reference: [Gamespy Presence SDK](#)

See Also: [GPCallback](#), [gpCheckUser](#), [GPResult](#)

GPConnectResponseArg

The arg parameter passed through to a GP_CALLBACK call after attempting to connect is of this type.

```
typedef struct  
{  
    GPResult result;  
    GPProfile profile;  
    gsi_char uniquenick[GP_UNIQENICK_LEN];  
} GPConnectResponseArg;
```

Members

result

Result of the connection; GP_NO_ERROR if successful.

profile

Profile for the user being connected.

uniquenick

Uniquenick for the newly connected user.

Section Reference: [Gamespy Presence SDK](#)

See Also: [gpConnect](#), [GPCallback](#), [GPResult](#)

GPDeleteProfileResponseArg

This arg data type contains the data for a delete profile operation. It is generated by a call to the callback passed to `gpDeleteProfile`.

```
typedef struct  
{  
    GPRResult result;  
    GPProfile profile;  
} GPDeleteProfileResponseArg;
```

Members

result

Result of the operation.

profile

the profile that was deleted.

Section Reference: [Gamespy Presence SDK](#)

See Also: [gpDeleteProfile](#)

GPErrorArg

Contains information about an error which has occurred.

```
typedef struct  
{  
    GPResult result;  
    GPErrorCode errorCode;  
    gsi_char * errorString;  
    GPEnum fatal;  
} GPErrorArg;
```

Members

result

The result of a call to a GP function. GP_NO_ERROR if successful.

errorCode

The specific cause of the error.

errorString

Readable text string representation of the errorCode.

fatal

Either GP_FATAL or GP_NON_FATAL to indicate whether error is fatal.

Section Reference: [Gamespy Presence SDK](#)

See Also: [GPErrorCode](#), [GPCallback](#)

GPGetInfoResponseArg

The arg parameter passed to a callback generated by a call to gpGetInfo is of this type. The structure provides information about the specified profile.

typedef struct

```
{  
    GPResult result;  
    GPProfile profile;  
    gsi_char nick[GP_NICK_LEN];  
    gsi_char uniquenick[GP_UNIQUENICK_LEN];  
    gsi_char email[GP_EMAIL_LEN];  
    gsi_char firstname[GP_FIRSTNAME_LEN];  
    gsi_char lastname[GP_LASTNAME_LEN];  
    gsi_char homepage[GP_HOMEPAGE_LEN];  
    int icquin;  
    gsi_char zipcode[GP_ZIPCODE_LEN];  
    gsi_char countrycode[GP_COUNTRYCODE_LEN];  
    float longitude;  
    float latitude;  
    gsi_char place[GP_PLACE_LEN];  
    int birthday;  
    int birthmonth;  
    int birthyear;  
    GPEnum sex;  
    GPEnum publicmask;  
    gsi_char aimname[GP_AIMNAME_LEN];  
    int pic;  
    int occupationid;  
    int industryid;  
    int incomeid;  
    int marriedid;  
    int childcount;  
    int interests1;  
    int ownership1;  
    int conntypeid;  
} GPGetInfoResponseArg;
```

Members

result

The result of the inquiry; GP_NO_ERROR if successful.

profile

The profile for which the info was requested.

nick

The nick for this profile info.

uniquenick

The unquenick for this profile info.

email

The email for this profile info.

firstname

The firstname for this profile info.

lastname

The lastname for this profile info.

homepage

The homepage for this profile info.

icquin

The ICQ UIN(User Identification Number) for this profile info.

zipcode

The zipcode for this profile info.

countrycode

The countrycode for this profile info.

longitude

Negative is west, positive is east. (0, 0) means unknown.

latitude

Negative is south, positive is north. (0, 0) means unknown.

place

E.g., "USA|California|Irvine", "South Korea|Seoul", "Turkey".

birthday

The day part of this profile's birthday (1-31).

birthmonth

The month part of this profile's birthday (1-12).

birthyear

The year part of this profile's birthday.

sex

An enum indicating the sex for this profile info. The possible values are:

GP_MALE -- The sex for this profile info is male.

GP_FEMALE -- The sex for this profile info is female.

GP_PAT -- The sex for this profile info is unknown.

publicmask

publicmask

A bitwise-or of enums indicating which parts of this profile's info are public. If the value of publicmask is GP_MASK_NONE then no info is masked. If it is GP_MASK_ALL then all of the mask-able info is masked. If any of the following bits are set, then the corresponding info is masked. If the bit is not set, the info is public:

GP_MASK_HOMEPAGE

This profile's homepage info.

GP_MASK_ZIPCODE

This profile's zipcode info.

GP_MASK_COUNTRYCODE

This profile's countrycode info.

GP_MASK_BIRTHDAY

This profile's birthday info.

GP_MASK_SEX

This profile's sex info.

If info is masked, then its value in the structure should not be used.

For example, if the GP_MASK_BIRTHDAY bit is set, the birthday, birthmonth, and birthyear fields should not be accessed.

aimname

The AOL IM screen name for this profile info.

occupationid

The occupation id for this profile info.

industryid

The industry id for this profile info.

incomeid

The income for this profile info.

marriedid

The marital status for this profile info.

childcount

The child count for this profile info.

interests1

The interests for this profile info.

conntypeid

The connection type for this profile info.

Remarks

If result is not GP_NO_ERROR, then the operation did not complete successfully, and the rest of this structure is invalid and should not be accessed.

Section Reference: [Gamespy Presence SDK](#)

See Also: [GPCallback](#), [gpGetInfo](#)

GPGetReverseBuddiesResponseArg

The arg parameter passed to a callback generated by a call to gpGetReverseBuddies is of this type.

```
typedef struct  
{  
    GPRResult result;  
    int numProfiles;  
    GPProfileSearchMatch * profiles;  
} GPGetReverseBuddiesResponseArg;
```

Members

result

Result of the inquiry; GP_NO_ERROR if successful.

numProfiles

The number of profiles that have you on their buddy list.

profiles

The list of profiles found.

Section Reference: [Gamespy Presence SDK](#)

See Also: [gpGetReverseBuddies](#)

GPGetUserNicksResponseArg

The arg parameter passed to a callback generated by a call to gpGetUserNicks is of this type.

```
typedef struct  
{  
    GPRResult result;  
    gsi_char email[GP_EMAIL_LEN];  
    int numNicks;  
    gsi_char ** nicks;  
    gsi_char ** uniquenicks;  
} GPGetUserNicksResponseArg;
```

Members

result

The result of the inquiry; GP_NO_ERROR if successful.

email

The eMail address being inquired about.

numNicks

The number of profiles found to match the given eMail/password. If 0, then the email and password did not match. If you are unsure if the email address passed to gpGetUserNicks is valid, call gplsValidEmail first. Then a value of 0 numNicks will always mean that the email address was valid but the password was incorrect.

nicks

The list of profile Nicknames, numNicks in length.

uniquenicks

The list of profile Uniquenicks, numNicks in length.

Section Reference: [Gamespy Presence SDK](#)

GPIsValidEmailResponseArg

The arg parameter passed to a callback generated by a call to `gplsValidEmail` is of this type.

```
typedef struct  
{  
    GPResult result;  
    gsi_char email[GP_EMAIL_LEN];  
    GPEnum isValid;  
} GPIsValidEmailResponseArg;
```

Members

result

The result of the inquiry; GP_NO_ERROR if successful.

email

The eMail address being inquired about.

isValid

GPTrue if a user exists with the given eMail address; GPFalse otherwise.

Section Reference: [Gamespy Presence SDK](#)

GPNewProfileResponseArg

The arg parameter passed to a callback generated by a call to gpNewProfile is of this type.

```
typedef struct  
{  
    GPRResult result;  
    GProfile profile;  
} GPNewProfileResponseArg;
```

Members

result

The result of the inquiry; GP_NO_ERROR if successful.

profile

The newly created profile, if successful.

Section Reference: [Gamespy Presence SDK](#)

GPNewUserResponseArg

The arg parameter passed to a callback generated by a call to gpNewUser is of this type.

```
typedef struct  
{  
    GPRResult result;  
    GPProfile profile;  
} GPNewUserResponseArg;
```

Members

result

The result of the creation attempt; GP_NO_ERROR if successful.

profile

The profile created for the new user, if successful.

Section Reference: [Gamespy Presence SDK](#)

See Also: [gpNewUser](#)

GPProfileSearchMatch

Information about a profile which is returned by a requested search. Is often collected in a list, such as those found in GPGetReverseBuddiesResponseArg or GPProfileSearchResponseArg,.

typedef struct

```
{  
    GPProfile profile;  
    gsi_char nick[GP_NICK_LEN];  
    gsi_char uniquenick[GP_UNIQUENICK_LEN];  
    gsi_char firstname[GP_FIRSTNAME_LEN];  
    gsi_char lastname[GP_LASTNAME_LEN];  
    gsi_char email[GP_EMAIL_LEN];  
} GPProfileSearchMatch;
```

Members

profile

Object representing this matching profile.

nick

The profile's nickname.

uniquenick

The profile's unquenick.

firstname

The first name for the profile.

lastname

The last name for the profile.

email

The eMail address for the profile.

Section Reference: [Gamespy Presence SDK](#)

See Also: [GPGetReverseBuddiesResponseArg](#),
[GPProfileSearchResponseArg](#)

GPProfileSearchResponseArg

The arg parameter passed to a callback generated by a call to gpProfileSearch is of this type. Contains information about the profiles that matched the search criteria.

```
typedef struct  
{  
    GPRResult result;  
    int numMatches;  
    GPEnum more;  
    GPProfileSearchMatch * matches;  
} GPProfileSearchResponseArg;
```

Members

result

The result of the inquiry; GP_NO_ERROR if successful.

numMatches

The number of profiles in the *matches* list.

more

[In/Out] GP_MORE if there are more matches to come; GP_DONE if this is the last (or only) batch of matches.

matches

A list of information for the matching profiles.

Remarks

The callback for `gpProfileSearch` will only receive a limited number of results in each batch. If there are more results than those passed to the callback, the *more* member of the structure will be set to `GP_MORE`. If there are no more results, *more* will be set to `GP_DONE`.

The callback routine can leave *more* set to `GP_MORE`, which will tell the SDK to deliver the next batch of results, or it change *more* to `GP_DONE`, which will tell the SDK to stop delivering information.

Section Reference: [Gamespy Presence SDK](#)

See Also: [gpProfileSearch](#), [GPProfileSearchMatch](#)

GPRrecvBuddyMessageArg

Information sent to the GP_RECV_BUDDY_MESSAGE callback.

```
typedef struct  
{  
    ;  
    GProfile profile;  
    unsigned int date;  
} GPRrecvBuddyMessageArg;
```

Members

profile

Profile for the Buddy who sent the message.

date

Timestamp of the message.

Section Reference: [Gamespy Presence SDK](#)

See Also: [gpSetCallback](#), [GPEnum](#)

GPrecvBuddyRequestArg

Information sent to the GP_RECV_BUDDY_REQUEST callback.

```
typedef struct  
{  
    GProfile profile;  
    unsigned int date;  
    gsi_char reason[GP_REASON_LEN];  
} GPrecvBuddyRequestArg;
```

Members

profile

Profile of the buddy who has made the request.

date

The timestamp of the request.

reason

The reason for the request.

Section Reference: [Gamespy Presence SDK](#)

See Also: [gpSendBuddyRequest](#), [gpSetCallback](#), [GPEnum](#)

GPRRecvBuddyRevokeArg

Information sent to the GP_RECV_BUDDY_REVOKE callback.

```
typedef struct  
{  
    GPProfile profile;  
    unsigned int date;  
} GPRRecvBuddyRevokeArg;
```

Members

profile

Profile ID of the buddy.

date

Date when the action occurred.

Section Reference: [Gamespy Presence SDK](#)

See Also: [gpSetCallback](#), [gpRevokeBuddyAuthorization](#)

GPrecvBuddyStatusArg

Information sent to the GP_RECV_BUDDY_STATUS callback.

```
typedef struct  
{  
    GProfile profile;  
    unsigned int date;  
    int index;  
} GPrecvBuddyStatusArg;
```

Members

profile

This object represents the buddy whose status has changed.

date

The timestamp of the change.

index

This buddy's index in the buddy list. This index can be used in a call to `gpGetBuddyStatus` to get more information on the buddy's new status.

Section Reference: [Gamespy Presence SDK](#)

See Also: [gpSetStatus](#), [gpSetCallback](#), [GPEnum](#)

GPRecvGameInviteArg

Information sent to the GP_RECV_GAME_INVITE callback.

```
typedef struct  
{  
    GPProfile profile;  
    int productID;  
    gsi_char location[GP_LOCATION_STRING_LEN];  
} GPRecvGameInviteArg;
```

Members

profile

Profile of the buddy who the message is from.

productID

The product ID of the game to which the remote profile is inviting the local profile.

location

The location string for the invite.

Section Reference: [Gamespy Presence SDK](#)

See Also: [gpSetCallback](#), [GPCallback](#), [GPEnum](#)

GPRRegisterCdKeyResponseArg

The arg parameter passed to a callback generated by a call to `gpRegisterCdKey` is of this type.

```
typedef struct  
{  
    GPRResult result;  
} GPRRegisterCdKeyResponseArg;
```

Members

result

The result of the register uniquenick operation; GP_NO_ERROR if successful.

Section Reference: [Gamespy Presence SDK](#)

See Also: [gpRegisterCdKey](#), [GPResult](#)

GPRRegisterUniqueNickResponseArg

The arg parameter passed to a callback generated by a call to `gpRegisterUniqueNick` is of this type.

```
typedef struct  
{  
    GPRResult result;  
} GPRRegisterUniqueNickResponseArg;
```

Members

result

The result of the register uniquenick operation; GP_NO_ERROR if successful.

Section Reference: [Gamespy Presence SDK](#)

See Also: [gpRegisterUniqueNick](#), [GPResult](#)

GPSuggestUniqueNickResponseArg

The arg parameter passed to a callback generated by a call to gpSuggestUniqueNick is of this type.

```
typedef struct  
{  
    GPRResult result;  
    int numSuggestedNicks;  
    gsi_char ** suggestedNicks;  
} GPSuggestUniqueNickResponseArg;
```

Members

result

The result of the suggest uniquenick operation; GP_NO_ERROR if successful.

numSuggestedNicks

The number of suggested uniquenicks contained in this struct.

suggestedNicks

An array of suggested uniquenicks. The number of elements in the array is specified by numSuggestedNicks.

Section Reference: [Gamespy Presence SDK](#)

See Also: [gpSuggestUniqueNick](#), [GPResult](#)

GPTransferCallbackArg

The arg parameter passed to a Transfer Callback .

```
typedef struct  
{  
    GPTransfer transfer;  
    GPEnum type;  
    int index;  
    int num;  
    gsi_char * message;  
} GPTransferCallbackArg;
```

Members

transfer

The transfer object this callback is for.

type

The type of information being passed to the application. See the "Transfer callback type" section of GPEnum.

index

If this callback is related to a specific file being transferred, this is that file's index.

num

An integer used in conjunction with certain "type" values to pass supplementary information to the program.

message

If the type is GP_TRANSFER_SEND_REQUEST, GP_TRANSFER_ACCEPTED, or GP_TRANSFER_REJECTED, then this may point to a user-readable text message sent with the request or reply. The message will be invalid once this callback returns.

Remarks

The possible values for *type* are:

GP_TRANSFER_SEND_REQUEST

A remote profile wants to send files to the local profile. The transfer object for this callback is a new transfer object. The application must call either `gpAcceptTransfer` or `gpRejectTransfer` in response to this message. If `gpAcceptTransfer` is called the transfer will start, and the object will exist until `gpFreeTransfer` is called on it. If `gpRejectTransfer` is called, the object will be freed and should not be referenced again.

Only the receiver gets this callback.

num is set to the number of files in the transfer.

message is set to the message passed to `gpSendFiles`.

GP_TRANSFER_ACCEPTED

A transfer request has been accepted. The files will now be sent.

Only the sender gets this callback.

message is set to the message passed to `gpAcceptTransfer`.

GP_TRANSFER_REJECTED

A transfer request has been rejected. Call `gpFreeTransfer` to free the transfer object.

Only the sender gets this callback.

message is set to the message passed to `gpAcceptTransfer`.

GP_TRANSFER_NOT_ACCEPTING

The remote profile is not accepting file transfers. Call `gpFreeTransfer` to free the transfer object.

Only the sender gets this callback.

GP_TRANSFER_NO_CONNECTION

A direct connection with the remote profile could not be established, and the transfer has been terminated. Call `gpFreeTransfer` to free the transfer object.

Only the sender gets this callback.

GP_TRANSFER_DONE

The file transfer has finished successfully. Call `gpFreeTransfer` to free the transfer object.

Both the sender and receiver get this callback.

GP_TRANSFER_CANCELLED

The file transfer has been cancelled before completing. Call `gpFreeTransfer` to free the transfer object.

Both the sender and receiver get this callback.

GP_TRANSFER_LOST_CONNECTION

The direct connection with the remote profile has been lost, and the transfer has been terminated. Call `gpFreeTransfer` to free the transfer object.

Both the sender and receiver get this callback.

GP_TRANSFER_ERROR

There was an error during the transfer process, and the transfer has been terminated. Call `gpFreeTransfer` to free the transfer object.

Both the sender and receiver get this callback.

GP_TRANSFER_THROTTLE

NOTE: Throttling is not currently implemented.

Either the local profile or the remote profile has set the throttle.

Both the sender and receiver get this callback.

If positive, `num` is the throttle setting in bytes-per-second (Bps). A throttle of zero means a paused connection, and a throttle of `-1` means there is no throttling.

GP_FILE_BEGIN

A file is about to be transferred.

Both the sender and receiver get this callback.

GP_FILE_PROGRESS

File data has been either sent or received.

Both the sender and receiver get this callback.

`num` is set to the number of bytes of this file that have been transferred so far.

GP_FILE_END

A file has finished transferring successfully. Always called after a `GP_FILE_BEGIN` (with zero or more `GP_FILE_PROGRESS` calls in between).

Both the sender and receiver get this callback.

GP_FILE_DIRECTORY

The current "file" being transferred is a directory name. This is the only callback called for directories (i.e., no `GP_FILE_BEGIN` or `GP_FILE_END`).

Both the sender and receiver get this callback.

GP_FILE_SKIP

The current file is being skipped. This may arrive instead of a `GP_FILE_BEGIN`, or while a file is being transferred (after `GP_FILE_BEGIN`, but before `GP_FILE_END`).

Both the sender and receiver get this callback.

GP_FILE_FAILED

The current file being transferred has failed. This may arrive instead of a GP_FILE_BEGIN, or while a file is being transferred (after GP_FILE_BEGIN, but before GP_FILE_END).

Both the sender and receiver get this callback.

num indicates the cause of the error. The possible values are:

GP_FILE_READ_ERROR

The sender had an error reading the file.

GP_FILE_WRITE_ERROR

The receiver had an error writing the file.

GP_FILE_DATA_ERROR

The MD5 check of the data being transferred failed. Because the MD5 check happens after the files has finished transferring, only the receiver will get this callback. The sender will simply get a GP_FILE_END callback.

Section Reference: [Gamespy Presence SDK](#)

See Also: [gpSetCallback](#)

Presence and Messaging SDK Enumerations

[GPEnum](#)

Presence and Messaging SDK's general enum list that holds context specific values for a variety of purposes.

[GPErrorCode](#)

Error codes which occur in GP processing.

[GPResult](#)

Possible Results which can be returned from GP functions. Check individual function definitions to see possible results.

GPEnum

Presence and Messaging SDK's general enum list that holds context specific values for a variety of purposes.

typedef enum

{

// Callbacks

GP_ERROR = 0,
GP_RECV_BUDDY_REQUEST,
GP_RECV_BUDDY_STATUS,
GP_RECV_BUDDY_MESSAGE,
GP_RECV_GAME_INVITE,
GP_TRANSFER_CALLBACK,
GP_RECV_BUDDY_AUTH,

// Global States

GP_INFO_CACHING= 0x0100,
GP_SIMULATION,
GP_INFO_CACHING_BUDDY_ONLY,

// Blocking

GP_BLOCKING= 1,
GP_NON_BLOCKING= 0,

// Firewall

GP_FIREWALL= 1,
GP_NO_FIREWALL= 0,

// Check Cache

GP_CHECK_CACHE= 1,
GP_DONT_CHECK_CACHE= 0,

// Is Valid Email.

GP_VALID= 1,
GP_INVALID= 0,

// Fatal Error

GP_FATAL= 1,
GP_NON_FATAL= 0,

// Sex

GP_MALE= 0x0500,
GP_FEMALE,
GP_PAT,

// Profile Search

GP_MORE= 0x0600,
GP_DONE,

// Set Info

GP_NICK= 0x0700,
GP_UNIQENICK,
GP_EMAIL,
GP_PASSWORD,
GP_FIRSTNAME,
GP_LASTNAME,
GP_ICQUIN,
GP_HOMEPAGE,
GP_ZIPCODE,
GP_COUNTRYCODE,
GP_BIRTHDAY,
GP_SEX,
GP_CPUBRANDID,
GP_CPUSPEED,
GP_MEMORY,
GP_VIDEOCARD1STRING,
GP_VIDEOCARD1RAM,
GP_VIDEOCARD2STRING,
GP_VIDEOCARD2RAM,
GP_CONNECTIONID,
GP_CONNECTIONSPEED,
GP_HASNETWORK,
GP_OSSTRING,
GP_AIMNAME,
GP_PIC,
GP_OCCUPATIONID,

GP_INDUSTRYID,
GP_INCOMEID,
GP_MARRIEDID,
GP_CHILDCOUNT,
GP_INTERESTS1,

// New Profile

GP_REPLACE= 1,
GP_DONT_REPLACE= 0,

// Is Connected

GP_CONNECTED= 1,
GP_NOT_CONNECTED= 0,

// Public mask

GP_MASK_NONE= 0x00000000,
GP_MASK_HOMEPAGE= 0x00000001,
GP_MASK_ZIPCODE= 0x00000002,
GP_MASK_COUNTRYCODE= 0x00000004,
GP_MASK_BIRTHDAY= 0x00000008,
GP_MASK_SEX= 0x00000010,
GP_MASK_EMAIL= 0x00000020,
GP_MASK_ALL= 0xFFFFFFFF,

// Status

GP_OFFLINE= 0,
GP_ONLINE= 1,
GP_PLAYING= 2,
GP_STAGING= 3,
GP_CHATTING= 4,
GP_AWAY= 5,

// CPU Brand ID

GP_INTEL= 1,
GP_AMD,
GP_CYRIX,
GP_MOTOROLA,
GP_ALPHA,

// Connection ID

GP_MODEM= 1,
GP_ISDN,
GP_CABLEMODEM,
GP_DSL,
GP_SATELLITE,
GP_ETHERNET,
GP_WIRELESS,

// Transfer callback type (the transfer is ended when these types are received)

GP_TRANSFER_SEND_REQUEST,
GP_TRANSFER_ACCEPTED,
GP_TRANSFER_REJECTED,
GP_TRANSFER_NOT_ACCEPTING,
GP_TRANSFER_NO_CONNECTION,
GP_TRANSFER_DONE,
GP_TRANSFER_CANCELLED,
GP_TRANSFER_LOST_CONNECTION,
GP_TRANSFER_ERROR,
GP_TRANSFER_THROTTLE,
GP_FILE_BEGIN,
GP_FILE_PROGRESS,
GP_FILE_END,
GP_FILE_DIRECTORY,
GP_FILE_SKIP,
GP_FILE_FAILED,

// GP_FILE_FAILED error

GP_FILE_READ_ERROR= 0x900,
GP_FILE_WRITE_ERROR,
GP_FILE_DATA_ERROR,

// Transfer Side

GP_TRANSFER_SENDER= 0xA00,
GP_TRANSFER_RECEIVER,

// UTM send options

GP_DONT_ROUTE= 0xB00,

```
// Quiet mode flags  
GP_SILENCE_NONE= 0x00000000,  
GP_SILENCE_MESSAGES= 0x00000001,  
GP_SILENCE_UTMS= 0x00000002,  
GP_SILENCE_LIST= 0x00000004,  
GP_SILENCE_ALL= 0xFFFFFFFF  
} GPEnum;
```

Constants

GP_RECV_BUDDY_AUTH

Required to receive new style buddy authorizations.

GP_PAT

I'm afraid to ask.

GP_OFFLINE

User is offline (disconnected from the server).

GP_ONLINE

User is online (connected to the server).

GP_PLAYING

User is playing a game.

GP_STAGING

User in a staging area for a game.

GP_CHATTING

User is chatting.

GP_DONT_ROUTE

Only sends the message directly to the player.

GP_SILENCE_MESSAGES

Messages will be silenced.

GP_SILENCE_UTMS

UTMs will be silenced.

GP_SILENCE_LIST

Buddy List requests, authorizations and revokes will be silenced.

GP_SILENCE_ALL

All GP traffic will be silenced.

Section Reference: [Gamespy Presence SDK](#)

GPErrorCode

Error codes which occur in GP processing.

typedef enum

{

// General

GP_GENERAL = 0x0000,

GP_PARSE,

GP_NOT_LOGGED_IN,

GP_BAD_SESSKEY,

GP_DATABASE,

GP_NETWORK,

GP_FORCED_DISCONNECT,

GP_CONNECTION_CLOSED,

// Login

GP_LOGIN = 0x0100,

GP_LOGIN_TIMEOUT,

GP_LOGIN_BAD_NICK,

GP_LOGIN_BAD_EMAIL,

GP_LOGIN_BAD_PASSWORD,

GP_LOGIN_BAD_PROFILE,

GP_LOGIN_PROFILE_DELETED,

GP_LOGIN_CONNECTION_FAILED,

GP_LOGIN_SERVER_AUTH_FAILED,

GP_LOGIN_BAD_UNIQUENICK,

GP_LOGIN_BAD_PREAUTH,

// Newuser

GP_NEWUSER= 0x0200,

GP_NEWUSER_BAD_NICK,

GP_NEWUSER_BAD_PASSWORD,

GP_NEWUSER_UNIQUENICK_INVALID,

GP_NEWUSER_UNIQUENICK_INUSE,

// Update UI

GP_UPDATEUI= 0x0300,

GP_UPDATEUI_BAD_EMAIL,

// New Profile

GP_NEWPROFILE= 0x0400,
GP_NEWPROFILE_BAD_NICK,
GP_NEWPROFILE_BAD_OLD_NICK,

// Update Profile

GP_UPDATEPRO= 0x0500,
GP_UPDATEPRO_BAD_NICK,

// Add Buddy

GP_ADDBUDDY= 0x0600,
GP_ADDBUDDY_BAD_FROM,
GP_ADDBUDDY_BAD_NEW,
GP_ADDBUDDY_ALREADY_BUDDY,

// Auth Add

GP_AUTHADD= 0x0700,
GP_AUTHADD_BAD_FROM,
GP_AUTHADD_BAD_SIG,

// Status

GP_STATUS = 0x0800,

// Buddy Message

GP_BM= 0x0900,
GP_BM_NOT_BUDDY,

// Get Profile

GP_GETPROFILE= 0x0A00,
GP_GETPROFILE_BAD_PROFILE,

// Delete Buddy

GP_DELBUDDY= 0x0B00,
GP_DELBUDDY_NOT_BUDDY,

// Delete Profile

GP_DELPROFILE= 0x0C00,
GP_DELPROFILE_LAST_PROFILE,

```
// Search
GP_SEARCH= 0x0D00,
GP_SEARCH_CONNECTION_FAILED,
GP_SEARCH_TIMED_OUT,

// Check
GP_CHECK= 0x0E00,
GP_CHECK_BAD_EMAIL,
GP_CHECK_BAD_NICK,
GP_CHECK_BAD_PASSWORD,

>/ Revoke
GP_REVOKE= 0x0F00,
GP_REVOKE_NOT_BUDDY,

// Register unique nick
GP_REGISTERUNIQUENICK= 0x1000,
GP_REGISTERUNIQUENICK_TAKEN,
GP_REGISTERUNIQUENICK_RESERVED,
GP_REGISTERUNIQUENICK_BAD_NAMESPACE,

// Register cdkey
GP_REGISTERCDKEY= 0x1100,
GP_REGISTERCDKEY_BAD_KEY,
GP_REGISTERCDKEY_ALREADY_SET,
GP_REGISTERCDKEY_ALREADY_TAKEN,

// AddBlock
GP_ADDBLOCK= 0x1200,
GP_ADDBLOCK_ALREADY_BLOCKED,

//RemoveBlock
GP_REMOVEBLOCK= 0x1300,
GP_REMOVEBLOCK_NOT_BLOCKED
} GPErrorCode;
```

Constants

GP_GENERAL

There was an unknown error.

GP_PARSE

Unexpected data was received from the server.

GP_NOT_LOGGED_IN

The request cannot be processed because user has not logged in.

GP_BAD_SESSKEY

The request cannot be processed because of an invalid session key.

GP_DATABASE

There was a database error.

GP_NETWORK

There was an error connecting a socket.

GP_FORCED_DISCONNECT

This profile has been disconnected by another login.

GP_CONNECTION_CLOSED

The server has closed the connection.

GP_LOGIN

There was an error logging in.

GP_LOGIN_TIMEOUT

The login attempt timed out.

GP_LOGIN_BAD_NICK

The nickname provided is incorrect.

GP_LOGIN_BAD_EMAIL

The e-mail address provided is incorrect.

GP_LOGIN_BAD_PASSWORD

The password provided is incorrect.

GP_LOGIN_BAD_PROFILE

The profile provided is incorrect.

GP_LOGIN_PROFILE_DELETED

The profile has been deleted.

GP_LOGIN_CONNECTION_FAILED

The server has refused the connection.

GP_LOGIN_SERVER_AUTH_FAILED

Could not authenticate server.

GP_LOGIN_BAD_UNIQUENICK

The uniquenick provided is incorrect.

GP_LOGIN_BAD_PREAUTH

There was an error validating the pre-authentication.

GP_NEWUSER

There was an error creating a new user.

GP_NEWUSER_BAD_NICK

A profile with that nick already exists.

GP_NEWUSER_BAD_PASSWORD

The password does not match the email address.

GP_NEWUSER_UNIQUENICK_INVALID

The uniquenick is invalid.

GP_NEWUSER_UNIQUENICK_INUSE

The uniquenick is already in use.

GP_UPDATEUI

There was an error updating the user information.

GP_UPDATEUI_BAD_EMAIL

A user with the email address provided already exists.

GP_NEWPROFILE

There was an error creating a new profile.

GP_NEWPROFILE_BAD_NICK

The nickname to be replaced does not exist.

GP_NEWPROFILE_BAD_OLD_NICK

A profile with the nickname provided already exists.

GP_UPDATEPRO

There was an error updating the profile information.

GP_UPDATEPRO_BAD_NICK

A user with the nickname provided already exists.

GP_ADDBUDDY

There was an error adding a buddy.

GP_ADDBUDDY_BAD_FROM

The profile requesting to add a buddy is invalid.

GP_ADDBUDDY_BAD_NEW

The profile requested is invalid.

GP_ADDBUDDY_ALREADY_BUDDY

The profile requested is already a buddy.

GP_AUTHADD

There was an error authorizing an add buddy request.

GP_AUTHADD_BAD_FROM

The profile being authorized is invalid.

GP_AUTHADD_BAD_SIG

The signature for the authorization is invalid.

GP_STATUS

There was an error with the status string.

GP_BM

There was an error sending a buddy message.

GP_BM_NOT_BUDDY

The profile the message was to be sent to is not a buddy.

GP_GETPROFILE

There was an error getting profile info.

GP_GETPROFILE_BAD_PROFILE

The profile info was requested on is invalid.

GP_DELBUDDY

There was an error deleting the buddy.

GP_DELBUDDY_NOT_BUDDY

The buddy to be deleted is not a buddy.

GP_DELPROFILE

There was an error deleting the profile.

GP_DELPROFILE_LAST_PROFILE

The last profile cannot be deleted.

GP_SEARCH

There was an error searching for a profile.

GP_SEARCH_CONNECTION_FAILED

The search attempt failed to connect to the server.

GP_SEARCH_TIMED_OUT

The search timed out.

GP_CHECK

There was an error checking the user account.

GP_CHECK_BAD_EMAIL

No account exists with the provided e-mail address.

GP_CHECK_BAD_NICK

No such profile exists for the provided e-mail address.

GP_CHECK_BAD_PASSWORD

The password is incorrect.

GP_REVOKE

There was an error revoking the buddy.

GP_REVOKE_NOT_BUDDY

You are not a buddy of the profile.

GP_REGISTERUNIQUENICK

There was an error registering the uniquenick.

GP_REGISTERUNIQUENICK_TAKEN

The uniquenick is already taken.

GP_REGISTERUNIQUENICK_RESERVED

The uniquenick is reserved.

GP_REGISTERUNIQUENICK_BAD_NAMESPACE

Tried to register a nick with no namespace set.

GP_REGISTERCDKEY

There was an error registering the cdkey.

GP_REGISTERCDKEY_BAD_KEY

The cdkey is invalid.

GP_REGISTERCDKEY_ALREADY_SET

The profile has already been registered with a different cdkey.

GP_REGISTERCDKEY_ALREADY_TAKEN

The cdkey has already been registered to another profile.

GP_ADDBLOCK

There was an error adding the player to the blocked list.

GP_ADDBLOCK_ALREADY_BLOCKED

The profile specified is already blocked.

GP_REMOVEBLOCK

There was an error removing the player from the blocked list.

GP_REMOVEBLOCK_NOT_BLOCKED

The profile specified was not a member of the blocked list.

Section Reference: [Gamespy Presence SDK](#)

See Also: [gpGetErrorCode](#), [GPErrorArg](#)

GPResult

Possible Results which can be returned from GP functions. Check individual function definitions to see possible results.

```
typedef enum  
{  
    GP_NO_ERROR,  
    GP_MEMORY_ERROR,  
    GP_PARAMETER_ERROR,  
    GP_NETWORK_ERROR,  
    GP_SERVER_ERROR  
} GPResult;
```

Constants

GP_NO_ERROR

Success.

GP_MEMORY_ERROR

Error occurred as result of insufficient memory.

GP_PARAMETER_ERROR

A provided parameter is either null or has an invalid value.

GP_NETWORK_ERROR

An error occurred while reading or writing across the network.

GP_SERVER_ERROR

Problem encountered trying to connect to the server.

Section Reference: [Gamespy Presence SDK](#)

Query and Reporting SDK

Overview

The GameSpy Server Browsing and Matchmaking systems are based on a central master server that collects information about all of the available game servers on the Internet and delivers that information to clients - either in-game clients based on GameSpy's Server Browsing and Matchmaking Toolkits, or out-of-game clients such as GameSpy Arcade.

The GameSpy Query & Reporting 2 SDK is used to allow your game server to report itself to the GameSpy master server backend and provide information to game clients who query it.

The system works as follows:

- When a game server starts up, it initializes the Query and Reporting 2 SDK. The SDK begins sending heartbeats to the GameSpy Master Server. These heartbeats contain information about the game server and how to connect to it.
- The Master Server sends several queries to the game server to verify that it exists, and test for various types of firewall and NAT devices that might be in front of the server.
- Once the game server has been verified, it is added to the list of available servers for the game.
- In-game and out-of-game clients then query the master server for the list of available servers.
- The clients send a small query to each available server to determine latency as well as server information.
- The Query and Reporting 2 SDK processes these incoming queries from clients and converts them to callbacks into your game code to retrieve the requested information. This information is returned as key/value pairs to the client.
- The client selects a game server to play on and initiates the connection process. This connection may be done directly through the game code, or through a 3rd party handshaking server if the game server is behind a NAT/Firewall and supports our NAT

negotiation technology. See the appendix of this document for more details on NAT/Firewall support.

Implementation of the Query and Reporting 2 SDK is simple, and you can generally have your game reporting to our backend in a matter of minutes.

Before you begin implementing the SDK, it is important that you confirm that this is the correct SDK for your game, as other options exist. Select the option below that best describes your game, or [contact devsupport@gamespy.com](mailto:devsupport@gamespy.com) if you are unsure before proceeding.

Dedicated server game using the Server Browsing Toolkit for in-game server lists, or only using GameSpy Arcade

If your game follows the "dedicated server" model, where an individual (who is typically not playing in the game on the same machine) starts a server which is available for other users to connect to and play on, and that server is always running, then you should use the Query and Reporting 2 SDK to report that server to our Master Server backend.

Peer-to-Peer game using the Matchmaking Toolkit (Peer SDK) for in-game lobby-based matchmaking

Games using the Peer SDK for lobby-based matchmaking do not need to implement the Query and Reporting 2 SDK directly. The Peer SDK "wraps" the functionality of the QR2 SDK and you can use the API functions provided by Peer for all game listing functionality.

Peer-to-Peer game using only GameSpy Arcade for matchmaking

If your Peer-to-Peer game does not have any in-game matchmaking solution, then you may not need to implement the Query and Reporting 2 SDK at all. Implementation is only required if your game supports "late-entry" - that is, allows players to join a game in progress, even after the initial group of players has started playing. In this case, you should report the game using the Query and Reporting 2 SDK running on the game host to ensure that out-of-game clients can still see the game and join it.

The rest of this document presents a simple, step-by-step set of

instructions for implementing the Query and Reporting 2 SDK in your game.

File Manifest

The following files should be included with this package. If any of the files are missing, please [contact devsupport@gamespy.com](mailto:devsupport@gamespy.com).

File	Description
<i>qr2.c</i>	Query and Reporting 2 SDK source
<i>qr2.h</i> in your source	Query & Reporting 2 main header - include this
<i>gr2regkeys.c</i>	Global array of pre-defined key names
<i>gr2regkeys.h</i>	Defines for pre-defined keys
<i>gr2sample.c</i>	Sample "game" in C that uses the SDK
<i>qr2csample.dsp</i>	MSVC project file to build the C sample game

In addition, to build the SDK and samples, you will need to separately download the GameSpy "common code" package, which includes the shared SDK code used by this SDK and others.

When extracting this package, make sure you preserve the directory tree in order to assure that the code builds correctly.

Implementation

Step 1: Determine The Data To Report

Before you begin writing code, you should determine what data you want to report about your game server to clients when they query it. There are three types of data you can report:

Server Data

This is general information about the game in progress, for example - the map that is being played, the type of game, and any specific game settings that would be of interest to players before they joined.

Player Data

This is information about a specific player that is in the current game, for example - the player's name, their current score, what team they are on, and the latency to the game server.

Team Data

This is information about a specific team in the current game, for example - the name of the team and the team score. If your game does not support team play you do not need to report any team information.

Data is reported in a "key / value" format. That is, each piece of data that you want to report has a specific key name associated with it, and when that key is requested you need to return the appropriate value. Key names are short strings that generally describe what the key is. To indicate the difference between types of keys, player keys always end in an "_" character (such as "score_") and team keys always end in a "_t" (such as [score_t](#)). Server keys can end in anything other than an _ or _t.

GameSpy has a standard list of keys that you can choose to report for your game. You do not have to report all these keys - in fact, depending on your game type, many of the keys may not be applicable at all. If you have additional data you want to report that is not covered by the standard list of keys, you can register "custom" keys for your game and

report any data you want.

The list of standard keys, and descriptions of the data they commonly represent, is below:

hostname

a descriptive host-defined string (can include spaces) that identifies the server (e.g. "Joe's Game!")

gamever

a version specifier (e.g. 1.23)

hostport

the port that the game networking is running on and that the client should connect to. If the game shares a port with the Query and Reporting 2 SDK, you do not need to specify this.

mapname

the map name (either filename or descriptive name)

gametype

string which specifies the type of game, or the mod being played.

gamevariant

if the particular game type has multiple variants, you can report it using this key.

numplayers

numeric string, number of players on the server

numteams

numeric string, number of teams on the server

maxplayers

numeric string, max number of players for this server

gamemode

string which specifies what is going on in the game at that time.

Modes include:

openwaiting

game has not yet started and players can join

closedwaiting

game has not yet started and players cannot join

closedplaying

game is in progress, no joining allowed

openplaying

game is in progress, players may still join

openstaging / closedstaging

Use to report that the game is in staging mode (should generally not be used directly - the Peer SDK handles this automatically).

exiting

server is shutting down

teampplay

number which defines the type of teamplay in use, or 0 for no teamplay. Values > 0 are up to the developer

fraglimit

number of total kills or points before a level change or game restart

teamfraglimit

number of total kills or points for a team before a level change or game restart

timelimit

amount of total time before a level change or game restart occurs (generally in minutes)

timeelapsed

amount of time (in seconds) since the current level or game started

roundtime

amount of time before a round ends (for round based games)

roundelapsed

amount of time (in seconds) that the current round has been in progress

password

0 or not present if no password is required to join, 1 if password is required. Implementation of actual password protection is up to the game developer's network code.

groupid

(optional) If the server being hosted is part of a "group room" then it needs to report which groupid it is part of (as passed in on launch)

`player_`

a player name (may include spaces)

`score_`

numeric string that contains the score (kills/points) for a single player

`skill_`

a skill rating, if applicable, for a single player

`ping_`

the ping for a player (as measured between the player and the server)

`team_`

the team a player is on, either numeric or string

`deaths_`

number of deaths a player has had

`pid_`

The profileID number for a player (if logged in with the P&M SDK)

`team_t`

the name for a team

`score_t`

the score for a team

Keys are identified by a numeric [KeyID](#) in addition to their registered names. The Query and Reporting 2 SDK always refers to keys by their [KeyID](#). The [KeyIDs](#) for the standard keys can be found as defines in the *qr2regkeys.h* file.

When you register custom keys for your game, you will need to select your own KeyID values for them. The values 0-49 are reserved, and 50-253 are available for custom keys. When you have decided what custom keys (if any) you need, you should assign [KeyID](#) values to them from that range (generally with defines so that you can refer to them easily in your code). The actual values do not matter, but you'll need to make sure

you always refer to the same key by the same [KeyID](#).

When you've decided on the custom keys you want to report, you'll need to register them with the [qr2_register_key\(\)](#) function. Call this function for each custom key you want to register before initializing the SDK.

```
void qr2_register_key(int keyid, const char *key);
```

Simply pass in the keyID you've chosen and the key name for your custom key. Remember that player keys need to end in "_" and team keys need to end in "_t".

Step 2: Create The Callback Function

You need to create a C callback function for each of the 3 key types in order to return the values for those keys when queried by a client. The SDK will call these functions when a user queries the server to get the latest information.

The prototype for the server key callback (from *qr2.h*) is:

```
typedef void (*qr2_serverkeycallback_t)(int keyid, c
```

A Sample function would be:

```
void server_key_callback(int keyid, qr2_buffer_t out
{
//add value to the buffer here
}
```

[KeyID](#) is the id number of the key being requested. You will need to look up the value for that particular game using your game's internal structures, and then add the value to the output buffer.

To add the value, simply call the [qr2_buffer_add\(\)](#) or [qr2_buffer_add_int\(\)](#) function:

```
void qr2_buffer_add(qr2_buffer_t outbuf, const char  
void qr2_buffer_add_int(qr2_buffer_t outbuf, int val
```

`Userdata` is a pointer that you can set to whatever you want in the `qr2_init()` function (described below). Generally this is used to store an object pointer or a structure pointer to your game data.

You will also need callbacks for the player and team keys. The prototype for these callbacks is:

```
typedef void (*qr2_playerteamkeycallback_t)(int keyi
```

The extra index parameter specifies the team or player index being requested (0-based).

In addition to the key/value callbacks, there are three additional callback functions you will need to create.

The first is the player/team count callback, which the SDK uses to determine the current number of players or teams (so that it can enumerate through each player or team by index to retrieve a specific key).

The prototype for this callback function is:

```
typedef int (*qr2_countcallback_t)(qr2_key_type key
```

The `keytype` parameter will tell you whether the player or team count is being requested. Simply return the current count from the function. If you do not support teams, just return 0 for the team count.

The next callback you need to provide is the key list callback. The SDK uses this callback to determine the complete list of keys you support - both standard and custom keys.

The prototype for this callback function is:

```
typedef void (*qr2_keylistcallback_t)(qr2_key_type k
```

The `keytype` parameter indicates what type of keys are being requested - server, player, or team. You should only add keys of the specified type to the key buffer. Use the `qr2_keybuffer_add()` function to add each supported key to the keybuffer in turn.

```
void qr2_keybuffer_add(qr2_keybuffer_t keybuffer, in
```

The final callback you will want to provide is the "add error" callback, which is called when the Master Server indicates to the game server that there has been an error adding it to the list. Typically this will only occur if the game server is behind some type of firewall or proxy that is blocking external traffic, and the game does not support NAT negotiation technology.

This callback can also be called if no traffic is received from the master server for approximately 40 seconds after starting reporting. This indicates that the master server is either inaccessible due to network difficulties, is down for maintenance, or is being blocked by an aggressive firewall.

The prototype for this function is below. See the `qr2.h` header file for descriptions of each parameter.

```
typedef void (*qr2_adderrorcallback_t)(qr2_error_t e
```

Step 3: Initialize the SDK

Once you have created the callback functions, you need to initialize the SDK to create the socket needed for queries and heartbeats. This only needs to be done once, and should be done before the actual game starts.

Simply call the `qr2_init()` function to create the required sockets.

The prototype for `qr2_init` is:

```
qr2_error_t qr2_init(qr2_t *qrec, const char *ip,
int baseport, const char *gamename,
const char *secret_key,
                int ispublic, int natnegotiate,
                qr2_serverkeycallback_t server_key
                qr2_playerteamkeycallback_t player
                qr2_playerteamkeycallback_t team_k
                qr2_keylistcallback_t key_list_cal
                qr2_countcallback_t playerteam_cou
                qr2_adderrorcallback_t adderror_ca
                void *userdata);
```

The Query and Reporting 2 SDK can be instantiated multiple times if you are running more than one server in a single process. If you are going to use it this way, you'll need to pass in a pointer to a `qr_t` variable for the first parameter. The returned value should then be passed into other functions that have a `qrec` parameter so that the correct instance is used.

If you are running a single game server instance per process (as most games will), then you can **simply pass NULL for the `qrec` parameter in all of the functions.**

IP

An optional dotted IP address for use on multi-homed machines. If you specify IP as NULL, all IP addresses on the machine will be bound. If your game networking supports binding to user-specified IPs, you should make sure the same IP is bound by the Query and Reporting 2 SDK.

baseport

The UDP port that the SDK will attempt to bind to accept queries on. If baseport is unavailable, up to 100 ports above baseport will be scanned to find an open port.

In the highly unlikely event that none of those ports are available, `e_qrbinderror` will be returned. You can also pass in 0 for baseport, to have a port chosen automatically by the operating system. However, this will make it harder to test the server and scan for it

on a LAN, since you never know which query port it is using, and is generally not recommended.

`gamename`

The unique gamename that you were issued with your secret key.

`secret_key`

The key you were issued with the gamename. We recommend that you do not just pass in a static string to the function, as this will show up in the executable's string table. Instead, you should set each character in the string individually, as shown in the sample programs. On consoles this is not a concern.

`ispublic`

Determines whether the server is reported to the GameSpy Master Server. If `ispublic` is 0, the server will not be reported. However, it will still respond to queries that come from clients broadcasting on the same LAN, and thus is essentially a "LAN-only" server. You should generally let server operators choose whether a server is public or private.

`natnegotiate`

a flag that indicates whether your game supports NAT negotiation technology for hosting behind a NAT or firewall. See the appendix on NAT support for further information on this parameter.

`userdata`

An implementation specific pointer that is passed each time a callback function is called. Use this to pass data structures or object pointers to the callback functions.

The six `qr2_*` callback parameters are the callback functions you created in the previous step.

If `qr2_init()` is successful, it will return `e_qrnoerror (0)`, otherwise it will return one of the error codes described in `qr2.h`.

Step 4: Process Queries in Your Main Loop

Somewhere in your main program (or message) loop, you need to call the `qr2_think()` function so that the SDK can process any pending

server queries.

This function should be called once every 10-100ms. Server queries are used by clients to gauge latency, so slow query replies will make the servers look more lagged than they are.

Step 5: Send statechange Heartbeats Where Needed

Whenever the [gamemode](#) of your game changes (going from waiting to playing, open to closed, playing to exiting, etc) you should call [qr2_send_statechanged\(\)](#) to send a statechanged heartbeat to the master. What this does is trigger the master server to immediately update the status of the server, instead of waiting up to 60 seconds for the next standard heartbeat to go out. You should be very careful to not send statechanged heartbeats in a tight loop, as this has caused flooding problems in the past when developers put this function in the same place as [qr2_think](#).

Step 6: Cleanup The SDK When Done

When your server is shutting down, you should call the [qr2_shutdown\(\)](#) function to close the query socket and do any misc. clean up. A final heartbeat is also sent to the master server automatically, indicating that the game server is being shut down and should be de-listed.

Step 7: Testing

Once you have the SDK implemented, you are ready to test it. For initial testing, it is best to make sure you are running on a machine connected directly to the Internet, with a real, routable IP address and no firewall or proxy. Although NATs and Firewalls are supported (as described in the Appendix), it can be difficult to tell whether a server listing failure is due to a NAT/Firewall or an implementation problem, so testing without the NAT/Firewall to begin with is highly recommended.

Once you've started your server and it has initialized the Query and Reporting 2 SDK, you can check the [Development Master Server Web Page](#) to confirm that the server has been listed on our master server.

The page defaults to the "gmtest" [gamename](#). If you are testing under a different [gamename/key](#) that you were issued, make sure to input the correct [gamename](#) on the page. The page will return a list of all servers currently listed for that [gamename](#).

If your server does not appear on the page, here are some things to check:

- Does the *qr2csample* application work on the machine you are running your game server on (try both the gmtest [gamename/key](#) and your own)? If the *qr2csample* does not show up on the web page when running, you can be fairly sure the problem is not in your implementation - but somehow network related. Confirm that you do not have a firewall or NAT device that might be blocking incoming queries from the master server.
- If the *qr2csample* works, but your game does not (on the same machine), then double-check your implementation - make sure you are calling [qr2_init](#), and that it is not returning an error codes. Make sure you are calling [qr2_think](#) at regular intervals of 10-100ms. You can try setting a breakpoint in your callbacks to see if any are getting called.
- You can use the *querytest* program described below to query your server and see if any results are returned. If results are not returned, the problem is almost definitely in your implementation. If results are returned, then the problem is either network or backend related.

Once you've got your server listing on our master server (or if you are trying to diagnose problems with listing), you can use the *querytest* program included with this SDK to send a query directly to your server and have all the keys/values the server sends back printed out. Use this program to confirm that the server is reporting all the keys and values you are trying to report.

To run *querytest*, simply specify the IP of the machine the server is running on, and the query port the game is using (as passed to [qr2_init](#)).

For example:

```
C:\Querytest.exe 1.2.3.4 27900
```

If it reports that the query timed out, then your game may not be implementing the QR2 SDK correctly.

If the data scrolls by too fast, you may either want to pipe the results to "more" or to a file - for example:

```
C:\Querytest.exe 1.2.3.4 27900 | more  
C:\Querytest.exe 1.2.3.4 27900 > out.txt
```

Appendix: Migration from a Previous Version of Query and Reporting SDK

The Query and Reporting 2 SDK, and the new backend that supports it, offers a number of significant advantages over the previous Query and Reporting SDK:

- Game servers can be queried for specific keys, instead of entire groups of keys so that just the data needed by clients is returned.
- The new wire protocol uses less bandwidth by not sending key names when a specific list of keys is requested, and not repeating key names for each player or team.
- The updated API removes the need to check buffer sizes when building query responses, all of that is handled by the SDK automatically now.
- Response values no longer need to be escaped to remove "\" characters, as that is no longer used as a protocol delimiter. The only invalid character for response strings is NUL (0). This will allow easier use of alternate character encodings such as UTF8.
- Adds support for our new NAT Negotiation technology

If your game has already implemented the Query and Reporting SDK, you are not required to switch to the QR2 SDK unless you want to take advantage of the benefits it provides, or use the advanced features of the new Server Browsing SDK, which require QR2.

To convert your game to use Query and Reporting 2, follow these steps:

1. Implement the new QR2 callbacks, as described in step 1 of the above implementation guide. You can remove the 4 query callbacks used in the original SDK.
2. Replace your call to `qr_init` with a call to `qr2_init`, filling in the appropriate additional parameters.
3. Replace your calls to `qr_process_queries` or `qr_process_queries_no_heartbeat` with a call to `qr2_think`. Note that there are no longer two processing functions. Whether or not heartbeats is sent is determined by the `ispublic`

parameter of `qr2_init`.

4. If you share sockets with the QR2 SDK, you will need to replace your call to `qr_parse_query` with a call to `qr2_parse_query`. For the QR2 SDK, the queries will no longer start with the "\n" character. You can identify queries by the magic bytes given in the `qr2.h` file. You also no longer need to NUL-terminate the query data before passing it to `qr2_parse_query`.
5. Replace calls to `qr_send_statechanged` and `qr_shutdown` with the equivalent qr2 calls.
6. Calls to `qr_send_exiting` are no longer needed, as an "exiting" heartbeat is sent as part of `qr2_shutdown`.

Once you've converted all your code and gotten it to compile, follow the testing guidelines in step 7 above to confirm your new implementation.

UNICODE Support

The GameSpy SDKs support an optional UNICODE interface for widestring applications. To use this interface, first define the symbol [GSI_UNICODE](#). Then, use widestrings wherever ANSI strings were previously called for. When in doubt, please refer to the header files for specific function declarations.

Although the GameSpy SDK interfaces support UNICODE parameters, some items may be stripped of their extra UNICODE information. These items include: nickname, email address, and URL strings. You may pass in widestring values, but they will first be converted to their ANSI counterparts before transmission.

***Note:** When using UNICODE, make sure to call [qr2_internal_key_list_free](#) after [qr2_shutdown](#) in order to free the internal key list created for UNICODE support. Not doing this will lead to memory leaks.

Appendix: NAT and Firewall Support

One of the largest challenges in game networking today is the variety of network topologies in use by players in homes and offices around the world. Technologies created to help users set up home networks and allow corporations to protect their internal networks have not been designed with gaming applications in mind, especially Peer to Peer applications, where a user may act as a host for other players. As more users get broadband connections, and the number of multi-PC households increases, this will only become a larger problem.

Soon we will see another reason emerge for people to purchase these devices - broadband consoles. Since most console users with broadband will also have a PC at home, we can expect a large percentage of online console users to be using a NAT device. Because of the frustration this can cause users, GameSpy has taken an aggressive stance in making sure that users can host multiplayer games no matter what their network topology. Before discussing GameSpy's specific solutions in this area, some background on the technologies and terminology is required.

Connection Types

There are three primary ways a user may be connected to the Internet.

Direct Connection

A user is said to have a "direct" connection if their machine is assigned a single, globally routable IP address for the duration that they are online, that address is not shared with any other users, and no network hardware between the user and the Internet is filtering or dropping any traffic. Direct Connection does not necessarily mean broadband - in fact, most dial-up users are considered Direct Connections as they have a true, routable IP address assigned to them whenever they dial up (even AOL users).

NAT Proxy Connection

NAT (short for Network-Address-Translating) proxies are becoming a

more common way for users (and companies) to share a single IP address with multiple computers.

NAT proxies come in either software form (e.g. SyGate, Windows ICS, WinRoute) or hardware (e.g. LinkSys Broadband router, and others).

Computers "behind" the NAT have private IP addresses - e.g. 192.168.0.1 - that are not accessible on the public Internet. The NAT device itself has 1 (or more) public, routable IP addresses. When a computer behind the NAT sends outgoing data or makes an outgoing connection, the NAT "edits" the packet to change the origin address to the NAT IP address, instead of the private IP address. It then chooses a local port on the NAT and uses that as the "public" port for the packet, instead of the port on the private machine. The NAT keeps a table that maps the "public" port on the NAT to the private IP and port that the packet originated from. When the destination machine replies, it goes to the NAT IP address and the "public" port. The NAT machine then forwards the packet to the private machine that matches the mapping, again rewriting the packet headers in the process so that the private machine has no idea anything is in the middle.

Firewalled Connection

A Firewall is a network device that sits on the network between the Internet and the user and looks at all traffic going back and forth to determine which traffic to allow. Firewalls are most often used in corporate environments, but many users have deployed home-based "software" firewalls. Firewalls can be configured hundreds of different ways depending on the desired behavior, however the most common configurations will not allow any "unsolicited" traffic to machines behind the firewall. Only after the machine behind the firewall has contacted an outside machine is any traffic from that outside machine allowed past the firewall.

Some firewalls are much more strict - only allowing outgoing TCP traffic on pre-defined ports (such as web browsing) - any firewall configured this strictly will likely be incompatible with any game. Many firewalls operate in an "invisible" fashion - the users behind the firewalls have publicly routable addresses and have no way to determine that a firewall is

blocking traffic. Firewalls may also be combined with NAT devices to provide both sets of functionality.

A special note about software firewalls (such as Zone Alarm or Black Ice): Our experience has shown that this type of software can be very unreliable when used in combination with games and the type of networking that games employ. This is due to both limitations of the software and unexpected interaction with games (such as popping up an invisible dialog during a full-screen game, causing an apparent lock-up). We highly recommend that users disable or at least turn down the security settings on these software firewalls when playing games to avoid problems.

NATs and Firewalls can be further broken down into two categories: Promiscuous and Non-promiscuous.

With a promiscuous NAT or Firewall, when a user sends a packet from their IP and port to a remote machine, a mapping is created on the network device that allows any outside machine to send data back to that user via the mapped port - so once the mapping has occurred, and the mapped port is determined by an outside machine, all other clients can learn about it from the outside machine and connect directly to the protected machine.

A non-promiscuous device is more restrictive - it will only allow incoming data from the specific IP and port that the outgoing data was sent to. Data from any other machines will be dropped.

Most (but not all) NAT devices are promiscuous, and most firewalls are configured to be non-promiscuous.

Hosting Methods

Over the years, a variety of methods have been developed to allow machines behind a NAT or firewall to host services (such as a game server).

Port Mapping

The most basic method is to configure the NAT or firewall device to pass unsolicited incoming traffic to a protected machine automatically. This is typically known as setting up a "port mapping".

For example, if a game accepts Q&R queries on port 27000, and hosts player connections in port 28000, a user could configure their NAT device to pass all traffic directed to those ports on the NAT back to the private address of their machine.

This method has a number of drawbacks:

- the user must be technical enough to understand how to set up a port mapping on their device
- the user must have admin access to their device (typically not available in a corporate environment)
- the developer must carefully document all ports used by the game that the user must map
- only one machine behind the device can be set up for a specific mapping (except in the case of firewalls, where you can often open a port range for all protected machines).

The primary advantage of this method is that it works without any changes to the game networking and can work with all types of networking - TCP and UDP based.

DMZ Host

Some NAT devices have an option known as "DMZ Host," whereby a specific machine behind the NAT can be designated to receive all unsolicited incoming traffic. This removes the need to set up individual port mappings for each game. However, only one machine can be the DMZ Host at a time, so only a single machine can host any services. In addition, setting a machine as DMZ Host eliminates many of the security features that a NAT provides, since it allows outside users to connect to the machine on any port.

Shared Socket

This method, which has been supported by GameSpy for the past year, allows players behind a promiscuous NAT to host games without any modification to their device or network connection. When a user hosts a game, a heartbeat is sent to the GameSpy master server. The master server automatically determines the "mapped" port that the NAT allocated, and informs other clients about the public address and the mapped port.

Since the NAT is promiscuous, outside clients are able to connect directly to that address and port. The method is called "shared socket" because it requires all game networking to operate on a single UDP socket that is shared with the GameSpy Query & Reporting SDK, since only a single mapping is created. The disadvantages of this method are that it does not support all devices (and it's currently impossible to tell which devices it supports until a user tries and it does not work), and that it requires the game networking use a single UDP socket - which many do, but not all.

NAT Negotiation

Also known as "port guessing", this is the new method supported by the Query and Reporting 2 SDK, as well as the additional NAT Negotiation SDK and the 3rd generation GameSpy Master Server. It requires using a 3rd party server (the NAT Negotiation Server, run by GameSpy) to coordinate the connection between two clients. Both clients connect to the negotiation server, and it determines what port their NAT devices has mapped, and what the next likely port to be mapped will be. The clients use this information to "guess" a port to connect to on the remote address, and begin sending packets to each other. After a few seconds, if the guessing is successful, a UDP connection will be open directly between the clients, and the NAT Negotiation server will not need to pass any data between them.

This method allows clients even behind non-promiscuous device to connect with each other, as long as the device has a predicable port allocation pattern (which currently most devices do). The method also allows developers to use a separate UDP port for each client if desired, although using a single UDP port (and even a shared socket) is still supported. TCP is not supported, because the protocol is not compatible with the type of simultaneous connection being attempted here. There is

no known way to splice a TCP connection between NAT clients modifications to the operating system.

Your Options

All of the above methods are compatible with the GameSpy SDKs, and you are free to choose any of them. Below is a description of how you would implement each option, and what it will mean in terms of network device support.

Implementing NAT Negotiation

Supporting the NAT Negotiation option will allow you game to work as a host behind the widest range of NAT devices and firewalls. To enable this, you will need to incorporate the separate GameSpy NAT Negotiation SDK - see that SDK's documentation for details. In the Query and Reporting 2 SDK, you simply need to set the `natnegotiate` flag in `qr2_init` to 1, to indicate to our backend that you support this method.

Our backend will determine if the host is behind a NAT or firewall that is blocking traffic, and inform clients when they need to use the NAT Negotiation Server to coordinate a connection to the host. Your game must use UDP-only networking to support NAT Negotiation, although you may use multiple UDP sockets if needed. The GameSpy Transport 2 SDK is fully compatible with NAT Negotiation.

Implementing Shared Socket

The Shared Socket option allows your game to host behind any promiscuous NAT or firewall device. If a user attempts to host behind a non-promiscuous device, the GameSpy Master Server will detect this and send an error message (via the QR2 error callback) to indicate a hosting failure. Details on implementing the shared socket method are in the separate "Shared Socket Implementation" appendix. To support Shared Socket, your game must use a single UDP socket for all client networking. The GameSpy Transport 2 SDK is fully compatible with the Shared Socket method.

Implementing NAT Negotiation + Shared Socket

You can also choose to implement NAT Negotiation and Shared Sockets, by using the NAT Negotiation SDK and the instructions for using shared sockets in the "Shared Socket Implementation" appendix. While this will not necessarily allow any extra players to host that could not otherwise (since the NAT Negotiation SDK works with both promiscuous and non-promiscuous NATs), it does remove the requirement of using the NAT Negotiation server for clients that are behind a promiscuous NAT - likely a large percentage of NAT users. This means faster connections to clients for these servers, less connection overhead, and reduced bandwidth from the master server.

If your game already uses a single UDP socket for all networking or uses the GameSpy Transport 2 SDK, we highly recommend using both Shared Socket and NAT Negotiation for the broadest and most complete NAT and Firewall support.

No Special Implementation

If your networking is not UDP based, or you are not able to implement either NAT Negotiation or Shared Socket, you can choose to do no special implementation. For clients to host your game behind a NAT or firewall, they will need to set up port mapping or DMZ Host options. If they are unable to do this, they will only be able to play as clients, connecting to servers that are hosted on direct connections. You should make sure you thoroughly document this in your manual to avoid any confusion.

Appendix: Shared Socket Implementation

The shared socket method works by using an external 3rd party server to determine the "public" port that the NAT has mapped private IP and port to. In the case of the Query and Reporting 2 SDK, we just use the GameSpy master server. Clients get the list of servers from the master server, including the IP address and port. Because the IP address and port are from the NAT, and are mapped to the private IP and port, clients can connect and be passed through to the private host.

Note that for this to work, the private IP and port used to send heartbeats to the master server must be the same as the IP and port for the game networking. Otherwise outside clients would be able to query the game for information, but not be able to connect to the game port because the NAT would not have a mapping for it. This means the QR2 SDK and the game must share a single UDP port for ALL game networking. TCP will not work as a host behind a NAT, and using multiple ports is not allowed because of the mapping problem.

Some games are already designed to be networked in this manner - the Quake and Unreal Engine games are two examples. They use a single UDP port on the server for all incoming connections and game data, and do not spawn a UDP socket for each client. If you do not have any game networking, you should consider the GameSpy Transport 2 SDK, which fully supports the shared socket method.

Once you have your game networking set up in this manner, integration with the Query and Reporting SDK is simple.

Instead of calling `qr2_init`, call `qr2_init_socket` and pass in your UDP game socket. The Query and Reporting 2 SDK will then use this socket to send heartbeats and reply to incoming queries.

However, the SDK still considers the game as the "owner" of the socket, so it will not try to read any data off it. Your game will read UDP datagrams off it as normal, and will need to determine if the packet received is a game packet, or a packet designed for the Query and Reporting 2 SDK. This will probably be easy to determine based on the

structure of your game networking packets, but the easiest way to identify an incoming query is to check the two characters - valid queries will always start with the magic numbers defined in *qr2.h*. Once you've read the data off the socket and identified it as an incoming query, you'll need to call the [qr2_parse_query](#) function to parse the query and reply to it. Simply pass the data and length to the function.

Note that you should NOT report a [hostport](#) key\value in the Info Callback, since the hostport will be the same as the query port, and will be determined by the NAT.

You should continue to call [qr2_think](#) at regular intervals as documented above, since this function is used for sending out heartbeats and other maintenance. Also note that you can use the [qr2_init_socket](#) method for all clients - whether or not they are behind a NAT - for clients not behind a NAT it will work fine.

Query and Reporting 2 SDK Functions

qr2_buffer_add	Add a string or integer to the qr2 buffer. This is used when responding to a qr2 query callback.
qr2_buffer_add_int	Add a string or integer to the qr2 buffer. This is used when responding to a qr2 query callback.
qr2_init	Initialize the Query and Reporting 2 SDK.
qr2_init_socket	Initialize the Query and Reporting 2 SDK. Allows control over the qr2 socket object.
qr2_internal_key_list_free	Frees the internal key list that is constructed when in GSI_UNICODE mode.
qr2_keybuffer_add	Add a key identifier to the qr2_keybuffer_t. This is used when enumerating the supported list of keys.
qr2_parse_query	When using the shared socket method with qr2_init_socket, use this function to pass qr2

	messages to the qr2 sdk.
qr2_register_clientconnected_callback	Sets the function that will be triggered when a client has connected.
qr2_register_clientmessage_callback	Sets the function that will be triggered when a client message is received.
qr2_register_key	Register a key with the qr2 sdk. This tells the sdk that the application will report values for this key.
qr2_register_natneg_callback	Sets the function that will be triggered when a nat negotiation request is received.
qr2_register_publicaddress_callback	Sets the function that will be triggered when the local clients public address is received.
qr2_send_statechanged	Notify the GameSpy master server of a change in gamestate.
qr2_shutdown	Frees memory allocated by the qr2 sdk. This includes freeing user registered keys.
qr2_think	Allow the qr2 sdk to continue processing. Server queries can

only be processed during this call.

qr2_buffer_add

Add a string or integer to the qr2 buffer. This is used when responding to a qr2 query callback.

```
void qr2_buffer_add(  
    qr2_buffer_t outbuf,  
    const gsi_char * value );
```

Routine	Required Header	Distribution
qr2_buffer_add	<qr2.h>	SDKZIP

Parameters

outbuf

[in] Buffer to add the value to. This is obtained from the qr2callback.

value

[in] String or integer value to append to the buffer.

Remarks

The **qr2_buffer_add** function appends a string to the buffer. The **qr2_buffer_add_int** function appends an integer to the bufer. These buffers are used to construct responses to user queries and typically contain information pertaining to the game status.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
qr2_buffer_add	qr2_buffer_addA	qr2_buffer_addW

qr2_buffer_addW and **qr2_buffer_addA** are UNICODE and ANSI mapped versions of **qr2_buffer_add**. The arguments of **qr2_buffer_addA** are ANSI strings; those of **qr2_buffer_addW** are wide-character strings.

Section Reference: [Gamespy Query and Reporting 2 SDK](#)

See Also: [qr2_buffer_add_int](#)

qr2_buffer_add_int

Add a string or integer to the qr2 buffer. This is used when responding to a qr2 query callback.

```
void qr2_buffer_add_int(  
    qr2_buffer_t outbuf,  
    int value );
```

Routine	Required Header	Distribution
qr2_buffer_add_int	<qr2.h>	SDKZIP

Parameters

outbuf

[in] Buffer to add the value to. This is obtained from the qr2callback.

value

[in] String or integer value to append to the buffer.

Remarks

The `qr2_buffer_add` function appends a string to the buffer. The **`qr2_buffer_add_int`** function appends an integer to the bufer. These buffers are used to construct responses to user queries and typically contain information pertaining to the game status.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
qr2_buffer_add_int	qr2_buffer_add_intA	qr2_buffer_add_intW

qr2_buffer_add_intW and **qr2_buffer_add_intA** are UNICODE and ANSI mapped versions of **qr2_buffer_add_int**. The arguments of **qr2_buffer_add_intA** are ANSI strings; those of **qr2_buffer_add_intW** are wide-character strings.

Section Reference: [Gamespy Query and Reporting 2 SDK](#)

See Also: [qr2_buffer_add](#)

qr2_init

Initialize the Query and Reporting 2 SDK.

```
qr2_error_t qr2_init(  
    qr2_t * qrec,  
    const gsi_char * ip,  
    int baseport,  
    const gsi_char * gamename,  
    const gsi_char * secret_key,  
    int ispublic,  
    int natnegotiate,  
    qr2_serverkeycallback_t server_key_callback,  
    qr2_playerteamkeycallback_t player_key_callback,  
    qr2_playerteamkeycallback_t team_key_callback,  
    qr2_keylistcallback_t key_list_callback,  
    qr2_countcallback_t playerteam_count_callback,  
    qr2_adderrorcallback_t adderror_callback,  
    void * userdata );
```

Routine	Required Header	Distribution
qr2_init	<qr2.h>	SDKZIP

Return Value

This function returns `e_qrnoerror` for a successful result. Otherwise a valid `qr2_error_t` is returned.

Parameters

qrec

[out] The initialized QR2 SDK object.

ip

[in] Optional IP address to bind to; useful for multi-homed machines. Usually pass NULL.

baseport

[in] Port to accept queries on. See remarks.

gamename

[in] The gamename, assigned by GameSpy.

secret_key

[in] The secret key for the specified gamename, also assigned by GameSpy.

ispublic

[in] Set to 1 for an Internet listed server, 0 for a LAN only server.

natnegotiate

[in] Set to 1 to allow NAT-negotiated connections.

server_key_callback

[in] Callback that is triggered when server keys are requested.

player_key_callback

[in] Callback that is triggered when player keys are requested.

team_key_callback

[in] Callback that is triggered when team keys are requested.

key_list_callback

[in] Callback that is triggered when the key list is requested.

playerteam_count_callback

[in] Callback that is triggered when the number of teams is requested.

adderror_callback

[in] Callback that is triggered when there has been an error adding it to the list.

userdata

[in] Pointer to user data. This is optional and will be passed unmodified to the callback functions.

Remarks

The **qr2_init** function initializes the qr2 SDK. The baseport parameter specifies which local port should be used to accept queries on. If this port is in use, the next port value will be tried. The qr2 sdk will try up to NUM_PORTS_TO_TRYports. (Currently set at 100.).

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
qr2_init	qr2_initA	qr2_initW

qr2_initW and **qr2_initA** are UNICODE and ANSI mapped versions of **qr2_init**. The arguments of **qr2_initA** are ANSI strings; those of **qr2_initW** are wide-character strings.

Section Reference: [Gamespy Query and Reporting 2 SDK](#)

qr2_init_socket

Initialize the Query and Reporting 2 SDK. Allows control over the qr2 socket object.

```
qr2_error_t qr2_init_socket(  
    qr2_t * qrec,  
    SOCKET s,  
    int boundport,  
    const gsi_char * gamename,  
    const gsi_char * secret_key,  
    int ispublic,  
    int natnegotiate,  
    qr2_serverkeycallback_t server_key_callback,  
    qr2_playerteamkeycallback_t player_key_callback,  
    qr2_playerteamkeycallback_t team_key_callback,  
    qr2_keylistcallback_t key_list_callback,  
    qr2_countcallback_t playerteam_count_callback,  
    qr2_adderrorcallback_t adderror_callback,  
    void * userdata );
```

Routine	Required Header	Distribution
qr2_init_socket	<qr2.h>	SDKZIP

Return Value

This function returns `e_qrnoerror` for a successful result. Otherwise a valid `qr2_error_t` is returned.

Parameters

qrec

[out] The initialized QR2 SDK object.

s

[in] Socket to be used for query traffic. This socket must have already been initialized.

boundport

[in] The port that the socket was bound to. Chosen by the developer.

gamename

[in] The gamename, assigned by GameSpy.

secret_key

[in] The secret key for the specified gamename, also assigned by GameSpy.

ispublic

[in] Set to 1 for an internet listed server, 0 for a LAN only server.

natnegotiate

[in] Set to 1 to allow natnegotiated connections.

server_key_callback

[in] Callback that is triggered when server keys are requested.

player_key_callback

[in] Callback that is triggered when player keys are requested.

team_key_callback

[in] Callback that is triggered when team keys are requested.

key_list_callback

[in] Callback that is triggered when the key list is requested.

playerteam_count_callback

[in] Callback that is triggered when the number of teams is requested.

adderror_callback

[in] Callback that is triggered when there has been an error adding it to the list.

userdata

[in] Pointer to user data. This is optional and will be passed unmodified to the callback functions.

Remarks

The **qr2_init_socket** function initializes the qr2 SDK. Instead of creating its own internal socket, the qr2 sdk will use the passed in socket for all traffic. The developer is responsible for receiving on this socket and passing received qr2 messages to **qr2_parse_query**.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
qr2_init_socket	qr2_init_socketA	qr2_init_socketW

qr2_init_socketW and **qr2_init_socketA** are UNICODE and ANSI mapped versions of **qr2_init_socket**. The arguments of **qr2_init_socketA** are ANSI strings; those of **qr2_init_socketW** are wide-character strings.

Section Reference: [Gamespy Query and Reporting 2 SDK](#)

qr2_internal_key_list_free

Frees the internal key list that is constructed when in GSI_UNICODE mode.

void qr2_internal_key_list_free();

Routine	Required Header	Distribution
qr2_internal_key_list_free	<qr2.h>	SDKZIP

Remarks

Developers should call this manually after calling `qr2_shutdown` while in `GSI_UNICODE` mode.

Section Reference: [Gamespy Query and Reporting 2 SDK](#)

qr2_keybuffer_add

Add a key identifier to the `qr2_keybuffer_t`. This is used when enumerating the supported list of keys.

```
void qr2_keybuffer_add(  
    qr2_keybuffer_t keybuffer,  
    int keyid );
```

Routine	Required Header	Distribution
qr2_keybuffer_add	<qr2.h>	SDKZIP

Parameters

keybuffer

[in] Buffer to append the key ID to.

keyid

[in] The ID of the supported key. Add one ID for each key supported.

Remarks

The **qr2_keybuffer_add** function is used to when enumerating the locally supported list of keys. Add the appropriate id number for each key supported.

Section Reference: [Gamespy Query and Reporting 2 SDK](#)

qr2_parse_query

When using the shared socket method with `qr2_init_socket`, use this function to pass qr2 messages to the qr2 sdk.

```
void qr2_parse_query(  
    qr2_t qrec,  
    gsi_char * query,  
    int len,  
    struct sockaddr * sender );
```

Routine	Required Header	Distribution
qr2_parse_query	<qr2.h>	SDKZIP

Parameters

qrec

[in] Initialize QR2 SDK initialized with `qr2_init_socket`.

query

[in] The QR2 packet received on the socket. See remarks.

len

[in] The length of the QR2 packet.

sender

[in] The sender of the packet.

Remarks

The **qr2_parse_query** function should be used in the shared socket implementation on qr2. In this implementation, the developer is responsible for creating and receiving on the socket, and forwarding qr2 messages to the sdk. The qr2 messages may be identified by the packet header. QR1 packets begin with a single backslash '\ character, QR2 packets begin with the QR_MAGIC_1 character followed by the QR_MAGIC_2 character.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
qr2_parse_query	qr2_parse_queryA	qr2_parse_queryW

qr2_parse_queryW and **qr2_parse_queryA** are UNICODE and ANSI mapped versions of **qr2_parse_query**. The arguments of **qr2_parse_queryA** are ANSI strings; those of **qr2_parse_queryW** are wide-character strings.

Section Reference: [Gamespy Query and Reporting 2 SDK](#)

qr2_register_clientconnected_callback

Sets the function that will be triggered when a client has connected.

```
void qr2_register_clientconnected_callback(  
    qr2_t qrec,  
    qr2_clientconnectedcallback_t cccallback );
```

Routine	Required Header	Distribution
qr2_register_clientconnected_callback	<qr2.h>	SDKZIP

Parameters

qrec

[in] QR2 SDK initialized with `qr2_init`.

cccallback

[in] Function to be called when a client has connected.

Section Reference: [Gamespy Query and Reporting 2 SDK](#)

See Also: [qr2_init](#), [qr2_clientconnectedcallback_t](#)

qr2_register_clientmessage_callback

Sets the function that will be triggered when a client message is received.

```
void qr2_register_clientmessage_callback(  
    qr2_t qrec,  
    qr2_clientmessagecallback_t cmcallback );
```

Routine	Required Header	Distribution
qr2_register_clientmessage_callback	<qr2.h>	SDKZIP

Parameters

qrec

[in] QR2 SDK initialized with qr2_init.

cmcallback

[in] Function to be called when a client message is received.

Remarks

The **qr2_register_clientmessage_callback** function is used to set a function that will be triggered when a client message is received.

Section Reference: [Gamespy Query and Reporting 2 SDK](#)

See Also: [qr2_init](#), [qr2_clientmessagecallback_t](#)

qr2_register_key

Register a key with the qr2 sdk. This tells the sdk that the application will report values for this key.

```
void qr2_register_key(  
    int keyid,  
    const gsi_char * key );
```

Routine	Required Header	Distribution
qr2_register_key	<qr2.h>	SDKZIP

Parameters

keyid

[in] Id of the key. See remarks.

key

[in] Name of the key.

Remarks

The **qr2_register_key** function tell the qr2 sdk that it should report values for the specified key. Key IDs 0 through NUM_RESERVED_KEYS are reserved for common key names. Keys upward to MAX_REGISTERED_KEYS are available for custom use.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
qr2_register_key	qr2_register_keyA	qr2_register_keyW

qr2_register_keyW and **qr2_register_keyA** are UNICODE and ANSI mapped versions of **qr2_register_key**. The arguments of **qr2_register_keyA** are ANSI strings; those of **qr2_register_keyW** are wide-character strings.

Section Reference: [Gamespy Query and Reporting 2 SDK](#)

qr2_register_natneg_callback

Sets the function that will be triggered when a nat negotiation request is received.

```
void qr2_register_natneg_callback(  
    qr2_t qrec,  
    qr2_natnegcallback_t nncallback );
```

Routine	Required Header	Distribution
qr2_register_natneg_callback	<qr2.h>	SDKZIP

Parameters

qrec

[in] QR2 SDK initialized with qr2_init.

nncallback

[in] Function to be called when a nat negotiation request is received.

Section Reference: [Gamespy Query and Reporting 2 SDK](#)

qr2_register_publicaddress_callback

Sets the function that will be triggered when the local clients public address is received.

```
void qr2_register_publicaddress_callback(  
    qr2_t qrec,  
    qr2_publicaddresscallback_t pacallback );
```

Routine	Required Header	Distribution
qr2_register_publicaddress_callback	<qr2.h>	SDKZIP

Parameters

qrec

[in] QR2 SDK initialized with qr2_init.

pacallback

[in] Function to be called when the local clients public address is received.

Section Reference: [Gamespy Query and Reporting 2 SDK](#)

qr2_send_statechanged

Notify the GameSpy master server of a change in gamestate.

```
void qr2_send_statechanged(  
    qr2_t qrec );
```

Routine	Required Header	Distribution
qr2_send_statechanged	<qr2.h>	SDKZIP

Parameters

qrec

[in] Initialized QR2 SDK object.

Remarks

The **qr2_send_statechanged** function notifies the GameSpy backend of a change in game state. This call is typically reserved for major changes such as mapname or gametype. Only one statechange message may be sent per 10 second interval. If a statechange is requested within this timeframe, it will be automatically delayed once the 10 second interval has elapsed.

Section Reference: [Gamespy Query and Reporting 2 SDK](#)

qr2_shutdown

Frees memory allocated by the qr2 sdk. This includes freeing user registered keys.

```
void qr2_shutdown(  
    qr2_t qrec );
```

Routine	Required Header	Distribution
qr2_shutdown	<qr2.h>	SDKZIP

Parameters

qrec

[in] QR2 SDK initialized with qr2_init.

Remarks

The **qr2_shutdown** function may be used to free memory allocated by the qr2 sdk. The qr2 sdk should not be used after this call. This call will cease server reporting and remove the server from the backend list.

Section Reference: [Gamespy Query and Reporting 2 SDK](#)

qr2_think

Allow the qr2 sdk to continue processing. Server queries can only be processed during this call.

```
void qr2_think(  
    qr2_t qrec );
```

Routine	Required Header	Distribution
qr2_think	<qr2.h>	SDKZIP

Parameters

qrec

[in] The initialized QR2 SDK.

Remarks

The **qr2_think** function allows the qr2 sdk to continue processing. This processing includes responding to user queries and triggering local callbacks. If q2_think is not called often, server responses may be delayed thereby increasing perceived latency. We recommend that **qr2_think** be called as frequently as possible. (10-15ms is not unusual.).

Section Reference: [Gamespy Query and Reporting 2 SDK](#)

Query and Reporting 2 SDK Callbacks

qr2_adderrorcallback_t	he callbacks provided to qr2_init; called in response to a message from the master server indicating a problem listing the server.
qr2_clientconnectedcallback_t	This callback is set via qr2_register_clientconnected_callback; called when a client has connected to the server.
qr2_clientmessagecallback_t	This callback is set via qr2_register_clientmessage_callback; called when a client message is received.
qr2_countcallback_t	One of the callbacks provided to qr2_init; called when the SDK needs to get a count of player or teams on the server.
qr2_keylistcallback_t	One of the callbacks provided to qr2_init; called when the SDK needs to determine all of the keys you game has values for.
qr2_natnegcallback_t	This callback is set via qr2_register_natneg_callback; called when a nat negotiation request is received.
qr2_playerteamkeycallback_t	One of the callbacks provided to qr2_init; called when a client requests information

about a player key or a team key.

[qr2_publicaddresscallback_t](#)

This callback is set via `qr2_register_publicaddress_callback`; called when the local client's public address is received.

[qr2_serverkeycallback_t](#)

One of the callbacks provided to `qr2_init`, called when a client requests information about a specific server key.

qr2_adderrorcallback_t

he callbacks provided to qr2_init; called in response to a message from the master server indicating a problem listing the server.

```
typedef void (*qr2_adderrorcallback_t)(  
    qr2_error_t error,  
    gsi_char *errmsg,  
    void *userdata );
```

Routine	Required Header	Distribution
qr2_adderrorcallback_t	<qr2.h>	SDKZIP

Parameters

error

[in] The code that can be used to determine the specific listing error.

errmsg

[in] A human-readable error string returned from the master server.

userdata

[in] The userdata that was passed into qr2_init.

Remarks

The most common error that will be reported is if the master is unable to list the server due to a firewall or proxy that would block incoming game packets.

Section Reference: [Gamespy Query and Reporting 2 SDK](#)

See Also: [qr2_init](#)

qr2_clientconnectedcallback_t

This callback is set via `qr2_register_clientconnected_callback`; called when a client has connected to the server.

```
typedef void (*qr2_clientconnectedcallback_t)(  
    SOCKET gameSocket,  
    struct sockaddr_in * remoteaddr,  
    void * userdata );
```

Routine	Required Header	Distribution
qr2_clientconnectedcallback_t	<qr2.h>	SDKZIP

Parameters

gameSocket

[in] The socket on which the client connected

remoteaddr

[in] The client's address and port.

userdata

[in] The userdata that was passed into `qr2_init`.

Section Reference: [Gamespy Query and Reporting 2 SDK](#)

See Also: [qr2_init](#), [qr2_register_clientconnected_callback](#)

qr2_clientmessagecallback_t

This callback is set via `qr2_register_clientmessage_callback`; called when a client message is received.

```
typedef void (*qr2_clientmessagecallback_t)(  
    gsi_char * data,  
    int len,  
    void * userdata );
```

Routine	Required Header	Distribution
<code>qr2_clientmessagecallback_t</code>	<code><qr2.h></code>	SDKZIP

Parameters

data

[in] The buffer containing the message

len

[in] The length of the data buffer

userdata

[in] The userdata that was passed into `qr2_init`.

Section Reference: [Gamespy Query and Reporting 2 SDK](#)

See Also: [qr2_init](#), [qr2_register_clientmessage_callback](#)

qr2_countcallback_t

One of the callbacks provided to qr2_init; called when the SDK needs to get a count of player or teams on the server.

```
typedef int (*qr2_countcallback_t)(  
    qr2_key_type keytype,  
    void * userdata );
```

Routine	Required Header	Distribution
qr2_countcallback_t	<qr2.h>	SDKZIP

Return Value

The callback should return the count for either the player or team, as indicated.

Parameters

keytype

[in] Indicates whether the player or team count is being requested
(key_player or key_team)

userdata

[in] The same userdata that was passed into qr2_init.

Remarks

If your game does not support teams, return 0 for the count of teams.

Section Reference: [Gamespy Query and Reporting 2 SDK](#)

See Also: [qr2_init](#)

qr2_keylistcallback_t

One of the callbacks provided to qr2_init; called when the SDK needs to determine all of the keys your game has values for.

```
typedef void (*qr2_keylistcallback_t)(  
    qr2_key_type keytype,  
    qr2_keybuffer_t keybuffer,  
    void * userdata );
```

Routine	Required Header	Distribution
qr2_keylistcallback_t	<qr2.h>	SDKZIP

Parameters

keytype

[in] The type of keys being requested (server, player, team). You should only add keys of this type to the keybuffer.

keybuffer

[in] The structure that holds the list of keys. Use `qr2_keybuffer_add` to add a key to the buffer.

userdata

[in] The same userdata that was passed into `qr2_init`.

Section Reference: [Gamespy Query and Reporting 2 SDK](#)

See Also: [qr2_init](#), [qr2_keybuffer_add](#)

qr2_natnegcallback_t

This callback is set via `qr2_register_natneg_callback`; called when a nat negotiation request is received.

```
typedef void (*qr2_natnegcallback_t)(  
    int cookie,  
    void *userdata );
```

Routine	Required Header	Distribution
qr2_natnegcallback_t	<qr2.h>	SDKZIP

Parameters

cookie

[in] The cookie associated with the NAT Negotiation request.

userdata

[in] The userdata that was passed into `qr2_init`.

Section Reference: [Gamespy Query and Reporting 2 SDK](#)

See Also: [qr2_init](#), [qr2_register_natneg_callback](#)

qr2_playerteamkeycallback_t

One of the callbacks provided to qr2_init; called when a client requests information about a player key or a team key.

```
typedef void (*qr2_playerteamkeycallback_t)(  
    int keyid,  
    int index,  
    qr2_buffer_t outbuf,  
    void *userdata );
```

Routine	Required Header	Distribution
qr2_playerteamkeycallback_t	<qr2.h>	SDKZIP

Parameters

keyid

[in] The key being requested.

index

[in] The zero-based index of the player or team being requested.

outbuf

[in] The destination buffer for the value information. Use `qr2_buffer_add` to report the value.

userdata

[in] The same userdata that was passed into `qr2_init`. You can use this for an object or structure pointer if needed.

Remarks

As a player key callback, this is called when a client requests information about a specific key for a specific player.

As a team key callback, this is called when a client requests the value for a team key.

If you don't have a value for the provided keyid, you should add an empty ("") string to the buffer.

Section Reference: [Gamespy Query and Reporting 2 SDK](#)

See Also: [qr2_init](#), [qr2_buffer_add](#)

qr2_publicaddresscallback_t

This callback is set via `qr2_register_publicaddress_callback`; called when the local client's public address is received.

```
typedef void (*qr2_publicaddresscallback_t)(  
    unsigned int ip,  
    unsigned short port,  
    void * userdata );
```

Routine	Required Header	Distribution
<code>qr2_publicaddresscallback_t</code>	<code><qr2.h></code>	SDKZIP

Parameters

ip

[in] IP address in string form: xxx.xxx.xxx.xxx

port

[in] Port number

userdata

[in] The userdata that was passed into qr2_init.

Remarks

The address is that of the externalmost NAT or firewall device, and is determined by the GameSpy master server during the qr2_init process.

Section Reference: [Gamespy Query and Reporting 2 SDK](#)

See Also: [qr2_init](#), [qr2_register_publicaddress_callback](#)

qr2_serverkeycallback_t

One of the callbacks provided to qr2_init, called when a client requests information about a specific server key.

```
typedef void (*qr2_serverkeycallback_t)(  
    int keyid,  
    int index,  
    qr2_buffer_t outbuf,  
    void *userdata );
```

Routine	Required Header	Distribution
qr2_serverkeycallback_t	<qr2.h>	SDKZIP

Parameters

keyid

[in] The key being requested.

index

[in] The 0-based index of the player or team being requested.

outbuf

[in] The destination buffer for the value information. Use `qr2_buffer_add` to report the value.

userdata

[in] The same userdata that was passed into `qr2_init`.

Remarks

If you don't have a value for the provided keyid, you should add an empty ("") string to the buffer.

Section Reference: [Gamespy Query and Reporting 2 SDK](#)

See Also: [qr2_init](#), [qr2_buffer_add](#)

Query and Reporting 2 SDK Enumerations

[qr2_error_t](#)

Constants returned from qr2_init and the error callback to signal an error condition.

[qr2_key_type](#)

Keytype indicates the type of keys being referenced -- server, player, or team.

qr2_error_t

Constants returned from qr2_init and the error callback to signal an error condition.

typedef enum

```
{  
    e_qrnoerror,  
    e_qrwsocerror,  
    e_qrbinderror,  
    e_qrdnserror,  
    e_qrconnerror,  
    e_qrnochallengeerror  
} qr2_error_t;
```

Constants

e_qrnoerror

No error occurred.

e_qrsockerror

A standard socket call failed, e.g. exhausted resources.

e_qrbinderror

The SDK was unable to find an available port to bind on.

e_qrdnserror

A DNS lookup (for the master server) failed.

e_qrconnerror

The server is behind a NAT and does not support negotiation.

e_qrnochallengeerror

No challenge was received from the master - either the master is down, or a firewall is blocking UDP.

Section Reference: [Gamespy Query and Reporting 2 SDK](#)

qr2_key_type

Keytype indicates the type of keys being referenced -- server, player, or team.

```
typedef enum  
{  
    key_server,  
    key_player,  
    key_team  
} qr2_key_type;
```

Constants

key_server

General information about the game in progress.

key_player

Information about a specific player.

key_team

Information about a specific team.

Section Reference: [Gamespy Query and Reporting 2 SDK](#)

SAKE Persistent Storage SDK

Overview

Sake (pronounced sah-keh, like the Japanese rice wine) is a GameSpy.net service which provides for flexible storage of arbitrary data on the GameSpy backend. This data can be global or player-specific, and it can be accessed or updated by game clients using the Sake SDK. Player-specific data can be private to a specific user, or it can be publicly accessible by all players. A database schema, created by the developer through a webpage interface, is used to organize the data. A range of data types can be stored in the database, including integers, floats, strings, dates and times, and files. Sake is simple to use, however it is a powerful system that can be used to provide game's with a whole new range of functionality.

A developer using Sake will deal with two separate components: the Sake Administration website and the Sake SDK. The Sake Administration site is used to setup the database schema which will store the game's data. The Sake SDK is used by the game to access the database.

File Manifest

The following files should be included with this package. If any of the files are missing, please [contact devsupport@gamespy.com](mailto:devsupport@gamespy.com).

File	Description
<i>sake.h</i>	GameSpy Sake header (all user functions are prototyped here)
<i>sakeMain.c</i>	Entry point for all user Sake functions
<i>sakeMain.h</i>	Common header for internal code
<i>sakeRequest.c</i>	Code handles and processes the Sake requests
<i>sakeRequest.h</i>	Header for Sake request-handling functions
<i>sakeRequestInternal.h</i>	Another header for Sake request-handling functions
<i>sakeRequestMisc.c</i>	Code to do internal processing of misc Sake requests
<i>sakeRequestModify.c</i>	Code to do internal processing of Sake requests that modify data
<i>sakeRequestRead.h</i>	Code to do internal processing of Sake requests that read data
<i>../common/gsoap.c</i>	GameSpy Soap code for XML streams
<i>../common/gsoap.h</i>	Header for GameSpy Soap code
<i>../common/gsXML.c</i>	Code that does XML reading/writing
<i>../common/gsXML.h</i>	Header for GameSpy XML code
<i>../ghttp/</i>	HTTP SDK
<i>/saketest/ ANSI C</i>	A Sake test app written for the command-line in ANSI C

Database

The core of Sake is the database schema used to store a game's data on the backend. A developer sets up the schema for a particular game through the Sake Administration website. The game can then access the database defined by this schema, using the Sake SDK.

The database for each game consists of a set of developer-created tables. Each entry in a table is referred to as a record. Each table has a set of fields, each of which describes a piece of data that will be stored in each of that table's records. In SQL terminology, a field is like a column and a record is like a row.

Example "high_scores" Database Table

recordid	ownerid	level	score
1	7623458	10	514
2	7821536	8	456
3	6998135	23	2678
4	7245991	36	4513
5	7400268	22	2449
6	6701102	10	498

Tables

In Sake, each table is uniquely identified within each game by a short string, called the tableid. For example, the tableid for the above sample table could be "high_scores". Every table contains some number of records, along with a set of fields which defines what values are stored in each record. A table is typically only accessible by a single game; however there is a mechanism on the backend which allows for a table to be shared across two or more games. This could be used to store per-player information that is shared by all of the games in a series or franchise. Developers create and manage tables using the Sake Administration website, and then they can access the data stored in those tables using the Sake SDK. In addition to the tableid, there are a few other important table properties which can be setup using the Sake

Administration website.

Owner Type

The owner type is used to designate if each individual table records is owned by a profile (which would identify a particular player), or if all of the records in a table are owned by the backend. If the owner type is set to **profile**, then each record contains a value which identifies the profile that owns that record. The owner would be the profile for which the record was originally created. The profile owner type is used for tables which store per-player information. If the owner type is set to **backend**, then the backend owns all of the records in the table. This would typically be used to store general global information which the developer may wish to periodically update. The owner type is a basic table property which must be set when a table is created - it cannot be changed after table creation

Permissions

Table permissions control the ability to create, read, update, or delete records. There are four permissions, *public create*, *public read*, *owner update*, and *owner delete*, each of which can be set to true or false:

- **public create** - if set to true, allows anyone to create records in the table. This will typically be set to true for tables with an owner type of profile, so that clients can create new records, and it will typically be set to false for tables with an owner type of backend.
- **public read** - if set to true, then anyone can read any record in the table. If the permission is set to false, then records can only be read by their owners. So this permission can be used to control if player-specific data stored in a table should be public or private.
- **owner update** - used to control whether or not a record can be updated by its' owner after it has been created. If the permission is set to true, then an owner can update the record any number of times. This may be used for a table where player preferences are stored. If the permission is set to false, then an owner can only create records, not update them. This may be used for a table in which records are used to record one-off events.
- **owner delete** - if set to true, then players can delete records which

they have created in the table. If, for example, records in the table correspond to items which the player has collected, then a record could be deleted if the player sells or loses an item. If the permission is set to false, then records cannot be deleted. This could be used in a table where each record stores a player's high scores, which would never be erased.

Rateable

The **rateable** option controls whether or not records in a table can be rated by users. For example, a racing game may allow all users to upload replay videos of their best races. Each video is stored in a record in a table. If that table's public read permission is set to true (giving other users access to those videos), and if the table's rateable option is set to true, then users can submit ratings for videos which they like or dislike. Other users can then see a video's average rating and the number of times it has been rated. The rating can also be used by the game to sort or filter a list of videos.

If a table has rateable set to true, then two fields are automatically added to that table. The **num_ratings** field stores the number of times that users have given that record a rating, and the **average_rating** field stores the average of all the ratings given to that record. See below for more information on fields. The maximum range for ratings is 0 through 255 (games can internally restrict that to a smaller range). Ratings are given as integers, however the average ratings is returned as a floating point number.

In addition, players can use the field name **my_rating** to obtain their personal rating on a given record. This can also be used when searching for records. So for example, if you wanted to only view records you have rated as > 100 then you would include "my_rating > 100" in the filter string. For records that the player has not yet rated, my_rating is set to -1 by default.

Limit Per Owner

The **limit per owner** option is used when users should be prevented from having more than a certain number of records in a table at any one time. If the option is set to 0, then users can have however many records they want. However if the option is set to some number greater than 0, then that number sets the maximum

number of records that any user can have in the table. For example, if a table is used to store player preferences, then the limit per owner for that table may be set to 1, since users would only ever have one set of preferences. Another example would be a table in which each record represents a special item that a player's character owns, and the game wants to limit each player to only 5 special items. Then the limit per owner for that table would be set to 5, which will prevent the user from storing more than 5 records in that table. If the user has 5 records in the table, then one of the five records will need to be deleted before adding a new record.

Fields

A field represents a piece of data which is stored in each record in a table. In the above example of a high scores table, there are four fields: recordid, ownerid, level, and score. Each record stored in the table has a value for each of these fields. All tables have one or more fields that are automatically created and managed by Sake. In the example, these are the recordid and ownerid fields. The developer can also add his own fields to any table. In the example, these are the level and score fields.

Developers add and manage fields using the Sake Administration website. A field consists of several pieces of information. First, each field has a name, such as "level", "score", or anything else. A field name must be unique within the table to which it belongs. A field also has a type, which defines what sort of data is stored in that field. Depending on the type, a field might also have a maximum length, and it might have a default value. The table below lists all of the possible field types, along with some information about them.

Sake Field Types

Type Name	Range	Has Default?	Has Max Length?	Comments
Byte	0 to 255	Yes	No	1 byte unsigned int
Short	-32,768 to	Yes	No	2 byte

	-32,767			unsigned int
Int	-2,147,483,648 to 2,147,483,647	Yes	No	4 byte unsigned int
Float	-1.79E308 to 1.79E308	Yes	No	8 byte floating point num
AsciiString	up to 1000 chars	Yes	Yes	String of single-byte chars
UnicodeString	up to 1000 chars	Yes	Yes	String of multi- byte chars
Boolean	true or false	Yes	No	
DateAndTime	1970 through 2038	Yes	No	Accurate to 1 second
BinaryData	up to 2000 bytes	No	Yes	Arbitrary binary data
FileID	n/a	No	No	References an uploaded file - treated as an int by the Sake SDK

Every table has a **recordid** field which is automatically included when that table is created. The field is an Int, and it is used to uniquely identify each record that is stored in the table. When a record is added to the table a value is automatically assigned to its recordid field by the backend. The recordid can then always be used to identify that record within the table.

If a table is created with its owner type property set to profile, then an **ownerid** field is automatically added to the table. The field is an Int, and it stores the profile which created the record. The value is filled in automatically by the backend when a record is created, and it never changes as long as that record exists. It is used by the backend to manage access to the record, and it can also be used by the game, through the Sake SDK, to figure out who created a record.

If a table has its `rateable` property set to `true`, then two additional fields are automatically added by the backend: **`num_ratings`** and **`average_rating`**. The `num_ratings` field is an `Int`, and it stores the number of times that a particular record has been given a rating by users. The `average_rating` field is a `Float`, and it stores the average of all the ratings that have been given to a record.

For a description of other special (non developer-defined) fields, see Appendix II below.

Records

While fields are used to define what sort of data will be stored in a table, the actual data is stored in entries known as records. Each record in a table contains a value for each field in that table. The value stored in the `recordid` field uniquely identifies a record within the table. Records are accessed using the Sake SDK. They can be created, updated, deleted, or read, depending on the permission properties for each table. More information can be found in the API section below.

Administration

A developer uses the Sake Administration website to configure a game's database schema. This is done by creating tables and then adding fields to those tables. Properties can also be configured for tables and fields. This schema is then used by the game through the Sake SDK. The website is located at: <http://tools.gamespy.net/SakeAdmin/>.

Starting

When you first visit the Sake Administration website, you will be asked to log in. Sake uses the GameSpyID system, which is the same login system used by the Presence & Messaging SDK (GP), GameSpy.com, GameSpy Arcade, FilePlanet.com, etc. Before using the Sake Administration website, you must be granted permission on the backend. To request permission for your GameSpyID account, send an email to devsupport@gamespy.com.

If you have permission to access your game's Sake Administration website, and you have logged in, then you will be able to start editing your game's database schema. You can do this by selecting your game from the main Game Selection page. This will lead you to the Game Tables page for your game. This page lists all of the tables for your game, and it also allows you to add new tables or edit existing tables.

Tables

The Game Tables page shows any tables that have been created for the selected game, along with each table's properties. The Table ID column shows the short string that uniquely identifies each table within the game's database. The Description column contains a developer supplied comment. This is only used on the Administration website and allows the developer to document the purpose of the table. The Owner Type column indicates if the table has an owner type of profile or backend. The Public Permissions and Owner Permissions columns show the settings for the public create, public read, owner update, and owner delete permissions. See the Database section above for more information about permissions. The Rateable column shows whether or not users can rate records

contained in the table. The Limit Per Owner column shows the maximum number of records that any user can have in the table at any one time. If the value is 0, then there is no limit.

There are several buttons to the left of each table. The Edit button can be used to edit that table's properties. All of the properties can be edited, aside from the owner type, which needs to be set when a table is created. When done editing, click Update to save the edit, or Cancel to cancel the edit. The Fields button brings you to a separate page which allows you to edit the list of fields for that table (see below for more information). The Reset button is used to delete all of the records from that table. The Delete button is used to remove a table from the list of table's associated with a game. Note that this won't actually delete the table and its records, but the table will no longer show up in the list, and it will no longer be accessible through the Sake SDK.

The Add a New Table box at the bottom of the page can be used to create a new table. The tableid and the owner type must be specified. All other properties can be set after the table has been created. The tableid can also be changed after the table is created, however the owner type cannot be changed. After entering the tableid and owner type, click the Create Table button to add the table to the list of table's for the current game.

Fields

To view and edit the list of fields for a particular table, click the Fields button to the left of that table on the Game Tables page. This will open up the Game Table Fields page for the selected table. You will see a list containing each of the fields in the table. In addition to any developer created fields, there will be one or more fields that were automatically created by Sake. Every table has a recordid field, which uniquely identifies each record within the table. If a table has an owner type of profile, then there will also be an ownerid field which contains the profile ID of the user that created that record. If a table has its rateable property set to true, then there will two additional fields, num_ratings and average_rating, which are described above.

The Name column shows the field's name, which is unique within the

table. The Description column contains a developer supplied comment. This is only used on the Administration website and allows the developer to document the purpose of the field. The Type column shows the type of data that is stored in the field. The database section above has a list of all the types. The Max Length column shows the maximum number of characters for AsciiString and UnicodeString fields and the maximum number of bytes for BinaryData fields. The Default column shows the default value, if the type for that field supports a default value.

There are two buttons to the left of most fields in the list. The Edit button is used to edit that field's properties. The field's name and description can always be edited. The max length and default value can also be edited, if the field's type uses those properties. A field's type cannot be edited after it is created. The Delete button is used to delete that field from the table. The recordid and ownerid fields cannot be edited or deleted. The num_ratings and average_rating fields also cannot be edited or deleted on the Game Table Fields page; however the fields can be removed by setting the table's rateable property to false.

The Add a New Field box at the bottom of the Game Table Fields page is used to add new fields to the current table. First enter a name for the new field, and then select a type from the dropdown box. Enter a max length and/or a default value depending on which type you selected. See the list of types above to see if either is needed. The name, max length, and default value can be changed after the field has been created, but the type can only be set during creation. When ready, click the Add field button to create the field and add it to the list. Once a field has been created, you can enter a description for it in the list.

SDK Implementation

Requirements

As with all GameSpy SDKs, Sake uses the GameSpy Common code. It also relies on the GameSpy HTTP SDK, which it uses to send requests to the Sake backend. The GameSpy Presence and Message SDK (GP) is also needed to provide authentication information for players.

Before using Sake, a game must have first performed the standard GameSpy Availability Check. This ensures that the GameSpy backend is available, and that the current game has access to the backend. See the Sake test app for sample code.

Sake uses the GameSpy Core object, which is part of the Common code, to manage tasks. The game must initialize the Core before using Sake. This is done by calling `gsCoreInitialize`. In order to allow Sake to process its requests, the core object must be periodically processed by calling `gsCoreThink`. When the game has finished using Sake it should shutdown the core with `gsCoreShutdown`. See the Sake test app for sample code for calling these functions.

```
void gsCoreInitialize();  
void gsCoreThink(gsi_time theMs);  
void gsCoreShutdown();
```

Sake needs the GP SDK to provide authentication information for players. This means that for a game to use Sake, it must also use GP. The player must successfully login with GP before using Sake, so that Sake can have access to GP's authentication information.

Field Types

The Sake SDK uses a few basic types to store data regarding fields. To represent a field itself, `SAKEField` is used.

```
typedef struct
```

```

{
    char          *mName;
    SAKEFieldType mType;
    SAKEValue     mValue;
} SAKEField;

```

A [SAKEField](#) object stores the field's name, the type of data stored in the field, and the value stored in the field. [SAKEFieldType](#) is used to indicate the type of data stored in a field.

```

typedef enum
{
    SAKEFieldType_BYTE,
    SAKEFieldType_SHORT,
    SAKEFieldType_INT,
    SAKEFieldType_FLOAT,
    SAKEFieldType_ASCII_STRING,
    SAKEFieldType_UNICODE_STRING,
    SAKEFieldType_BOOLEAN,
    SAKEFieldType_DATE_AND_TIME,
    SAKEFieldType_BINARY_DATA,

    SAKEFieldType_NUM_FIELD_TYPES
} SAKEFieldType;

```

It is important to note that all of the field types that can be created through the Administration site are represented here, with the exception of a **FileID**. That is because **FileIDs** must be handled specially on the backend, but from the perspective of the SDK they can be treated as **Ints**. So when reading a **FileID** field the backend will indicate it is a [SAKEFieldType_INT](#), and when updating a **FileID** field it should be updated as an [SAKEFieldType_INT](#).

The value for a field is stored in a [SAKEValue](#) union.

```

typedef union
{

```

```

    gsi_u8          mByte;
    gsi_i16         mShort;
    gsi_i32         mInt;
    float          mFloat;
    char           *mAsciiString;
    unsigned short *mUnicodeString;
    gsi_bool       mBoolean;
    time_t         mDateAndTime;
    SAKEBinaryData mBinaryData;
} SAKEValue;

```

The `mType` member of the `SAKEField` object to which this `SAKEValue` belongs is used to indicate which of the union members contains the actual value for this field. There is a union member corresponding to each of the types in the `SAKEFieldType` enum. `mByte`, `mShort`, `mInt`, and `mFloat` simply store integer or floating point values. `mAsciiString` and `mUnicodeString` contain pointers to strings which are NUL terminated for ASCII or double-NUL terminated for Unicode. To set the value of `mBoolean`, use `gsi_true` and `gsi_false`. However to check the value of `mBoolean`, the macros `gsi_is_true` and `gsi_is_false` should be used. `mDateAndTime` contains a date and time value stored in the same format as that returned by the standard `time()` function. `mBinaryData` contains arbitrary binary data stored in a `SAKEBinaryData` struct.

```

typedef struct
{
    gsi_u8 *mValue;
    int     mLength;
} SAKEBinaryData;

```

`mValue` points to the data itself, and `mLength` contains the number of bytes of data. `mValue` may be NULL if `mLength` is 0.

When a `SAKEField` is supplied to the SDK as part of an input object (described below under requests), then the game is responsible for providing the memory to which any pointers point. The field name and

any string or binary data pointers must point to memory which the game is managing. If a [SAKEField](#) object is passed to the SDK as an output object (described below under requests), then all the pointers will point to memory which the SDK is managing. This memory should not be freed, and any data which the game wants to access at a later point must be copied.

Startup and Cleanup

Before using Sake, the GameSpy Availability Check must have been performed and indicated that the game's backend is available, and the Core object must have been initialized, as described above. After these steps are completed, and the game is ready to start using Sake, it can call [sakeStartup](#).

```
SAKEStartupResult SAKE_CALL sakeStartup(SAKE *sakePt
```

The function returns a [SAKEStartupResult](#), which is an enumeration of possible results. If the result is [SAKEStartupResult_SUCCESS](#), then the startup has succeeded. Any other value indicates a failure, and the game should not continue calling other Sake functions.

The game supplies a pointer to a [SAKE](#) variable when calling [sakeStartup](#). If the startup is successful, then the variable will store a reference to the internal state of the Sake SDK. This [SAKE](#) reference is then used with most other calls to Sake functions. The reference is valid until the game shuts down the Sake SDK with [sakeShutdown](#).

```
void SAKE_CALL sakeShutdown(SAKE sake);
```

This shuts down the SDK and frees any memory that was allocated for the Sake object. After this function returns, the reference to the Sake object is no longer valid and should not be used.

After Sake has been shutdown, the game should shutdown the GameSpy Core object by calling [gsCoreShutdown](#). Sample code for this is available in the Sake test app.

Authentication

After Sake has been initialized, the game needs to provide authentication information which will identify the player and game. This allows the backend to ensure that the game can only access or modify information which the current player has permission to access or modify. There are two functions involved in authentication, and the game must call both of them before continuing with any other Sake usage. To set the game's authentication information, call `sakeSetGame`.

```
void SAKE_CALL sakeSetGame
(
    SAKE sake,
    const char *gameName,
    int gameId
);
```

The first parameter is the reference to the Sake object obtained when calling `sakeStartup`. The other two parameters are the gamename and gameId for the current game. These are provided on a per-game basis by GameSpy. If your game needs a gamename and gameId, or if you do not know the gamename or gameId for your game, contact devsupport@gamespy.com.

The function provides no indication of whether or not the gamename and gameId are correct. It only stores them with in the Sake object, and they are then passed along with any requests sent to the Sake backend. The backend will then check them and use the information to figure out which game's database is being used.

```
void SAKE_CALL sakeSetProfile
(
    SAKE sake,
    int profileId,
    const char *loginTicket
);
```

`sakeSetProfile` is used to provide authentication information for the

current player. The profile ID and login ticket are both obtained from the GameSpy Presence and Messaging SDK (GP). The profile ID uniquely identifies the current player to the backend, and the login ticket allows the backend to verify that the player is correctly identifying himself. Before calling [sakeSetProfile](#), the player should have successfully logged in using the GP SDK, which allows the GameSpy backend to authenticate the player.

The profile ID to pass to [sakeSetProfile](#) can be obtained in the callback that is called as a result of logging into GP. A [GPConnectResponseArg](#) struct is passed to the callback, and the struct has a member variable "profile" that stores the player's profile ID. While the player is logged on, the game should call the GP function [gpGetLoginTicket](#). This provides the login ticket which is then passed to [sakeSetProfile](#).

As with [sakeSetGame](#), [sakeSetProfile](#) provides no indication of whether or not the information provided is correct. It stores the profile ID and login ticket in the Sake object, and they are then passed along with any requests sent to the Sake backend. The backend checks them and uses them to authenticate the player and, for certain requests, identify which player's data is being access or updated.

Requests

To communicate with the Sake backend, the game sends requests through the Sake SDK. This is the primary functionality of the SDK. Once [sakeStartup](#) has been called, and the game has provided authentication information (see above), it can start sending requests. Requests allow the game to create records, update records, delete records, read records, and rate records, as well as check the record limit for a particular table (the limit per owner option set with the Administration website).

All of the request functions have a similar format. As an example, this is the function for a CreateRecord request.

```
SAKERequest SAKE_CALL sakeCreateRecord  
(
```

```
SAKE sake,  
SAKECreateRecordInput *input,  
SAKERequestCallback callback,  
void *userData  
);
```

All request functions take a reference to the sake object as the first parameter, a pointer to an input object as the second parameter, a reference to a callback as the third parameter, and a pointer to user data as the last parameter.

The type of the input object parameter is different for each request type - in this case the type is `SAKECreateRecordInput`. The input object contains the data that will be passed to the backend as part of the request. For a CreateRecord request, the input object contains the tableid of the table in which to create the record and the initial field values to store in the new record. An input object must be valid for the entire duration of a request, which it means it cannot be freed immediately after the request is initiated. It can only be freed if the request fails or after the request completes.

Request functions return a `SAKERequest` variable, which stores a reference to an internal object that tracks the request. If a request function returns a NULL value, then the request has failed to initialize. If that happens, `sakeGetStartRequestResult` can be called to get the reason for the failure.

```
SAKEStartRequestResult SAKE_CALL sakeGetStartRequest
```

It returns an enum value of type `SAKEStartRequestResult`, which will indicate the specific reason. It will always return the result for the most recent request that was attempted, so it must be called immediately after a failure to get the reason for that failure.

All request functions take a reference to a `SAKERequestCallback` as the third parameter.

```
typedef void (*SAKERequestCallback)
```

```
(
    SAKE sake,
    SAKERequest request,
    SAKERequestResult result,
    void *inputData,
    void *outputData,
    void *userData
);
```

If a request is started successfully, then callback will be called when the request completes. The first two parameters are references to the objects storing the Sake state and the request state. The third parameter is an enum value that indicates success or failure of the request.

[SAKERequestResult_SUCCESS](#) means success, any other value means failure. The fourth parameter is a pointer to the input object which was passed as the second parameter to the request. The fifth parameter is a pointer to an output object for this request, which will contain any data which the backend sent in response to the request. The specific types for these parameters depend on the type of request. For example, if the request was a [CreateRecord](#) request, then the types will be [SAKECreateRecordInput](#) and [SAKECreateRecordOutput](#), and the output object will store the recordid of the newly created record. Not all request types has output objects. If a request type does not have an output object, then `outputData` will be always be NULL when the callback is called. The final parameter is the same user data pointer that was passed to the request function.

The callback is where the game can see the result and any response to its request. If an input object was allocated dynamically, then the game can free that object from within the callback. However it is important to know that for certain request types, the output object may contain pointers to data stored in the input object. Therefore the input object should only be freed at the end of the function, after handling the output object. Also, the output object's data is only valid during the duration of the callback - it cannot be reference after the callback completes. So any data that needs to be accessed later must be copied before the callback returns.

The SDK can make the following requests, each of which follows the

format shown in the `sakeCreateRecord` request above. The only difference is the name of the request, and the Input struct used for each.:

```
//modifying Records
SAKERequest sakeCreateRecord(..., SAKECreateRecordIr
SAKERequest sakeUpdateRecord(..., SAKEUpdateRecordIr
SAKERequest sakeDeleteRecord(..., SAKEDeleteRecordIr

//retrieving Records
SAKERequest sakeSearchForRecords(..., SAKESearchForR
SAKERequest sakeGetMyRecords(..., SAKEGetMyRecordsIr
SAKERequest sakeGetRandomRecord(..., SAKEGetRandomRe
SAKERequest sakeGetSpecificRecords(..., SAKEGetSpeci

//miscellaneous
SAKERequest sakeRateRecord(..., SAKERateRecordInput
SAKERequest sakeGetRecordLimit(..., SAKEGetRecordLim
SAKERequest sakeGetRecordCount(..., SAKEGetRecordCou
```

Thinking

As described above in the Requirements section, the Sake SDK uses the GameSpy Core object to manage requests, so in order for requests to be processed the game must periodically call `gsCoreThink`.

```
void gsCoreThink(gsi_time theMS);
```

`gsCoreThink` tells the Core object to process any pending tasks. It takes one parameter, which tells it how long it can take to think. You will usually want to pass in 0 as the parameter, which will tell it to let each task do one round of processing.

Generally `gsCoreThink` will be called once each time a game runs through its main loop, or at a minimum every 50 milliseconds. It only needs to be called while Sake is in use, however calling it more often will not do any harm. It will just return without doing any processing if Sake is not in use. For sample code see the Sake test app.

File References

Sake supports storing files; however the files are not stored directly in the database. To store a file, the game must first upload it to the Sake File Server. If the file is uploaded successfully, the File Server will give the game a **fileid** which is used to uniquely identify that file, and which can then be stored directly in the Sake database as reference to that file.

The Sake backend keeps track of how many references to each file are stored in the database. If over a period of approximately 24 hours there are no references to a file, it will be deleted from the File server. This ensures that if a **fileid** is obtained from the database, it will still be valid for a period of time and available for download even if the last reference to that file was removed.

There is no way to overwrite an existing file. To change a file that is referenced from the database, first upload a new file, and then change the **fileid** in the database to reference the new file. If there are no other references to the old file, it will eventually be deleted.

File Uploading

[sakeGetFileUploadURL](#) is used to get a URL which can be used to upload files.

```
gsi_bool SAKE_CALL sakeGetFileUploadURL
(
    SAKE sake,
    gsi_char url[SAKE_MAX_URL_LENGTH]
);
```

The URL will identify both the game and the player that is uploading the file. After obtaining the URL, the file can be uploaded. We recommend using the GameSpy HTTP SDK, however other HTTP SDKs can be used. The file must be uploaded as an HTTP POST. See the Sake test app for a simple example of posting a file.

After completing the post, check the headers returned from the server for the result.

```
gsi_bool SAKE_CALL sakeGetFileResultFromHeaders
(
    const char *headers,
    SAKEFileResult *result
);
```

You can use [sakeGetFileResultFromHeaders](#) to do this automatically, or you can check the headers manually for the "Sake-File-Result" header. The value stored in the header is an integer, the possible values of which are enumerated in [SAKEFileResult](#).

[SAKEFileResult_SUCCESS](#) means that the file was uploaded successfully, while any other value indicates that there was an error uploading the file. Note that [sakeGetFileResultFromHeaders](#) returns [gsi_true](#) if it was able to find the "Sake-File-Result" header, and [gsi_false](#) if it was not able to find the header. So the return value does not in itself indicate that the file was uploaded successfully.

If the file was uploaded successfully, then the **fileid** that references the file can be obtained using [sakeGetFileIdFromHeaders](#).

```
gsi_bool SAKE_CALL sakeGetFileIdFromHeaders
(
    const char *headers,
    int *fileId
);
```

To get the fileid from the headers manually, look for the "Sake-File-Id" header. The fileid can now be stored in a **fileid** field in the database. If a file is uploaded, but the fileid is not stored in the database, then the file will be automatically deleted by the backend after approximately 24 hours.

File Downloading

A file can be downloaded once the **fileid** for that file has been obtained from the Sake database. [sakeGetFileDownloadURL](#) is used to get a download URL for a particular fileid.

```
gsi_bool SAKE_CALL sakeGetFileDownloadURL
(
    SAKE sake,
    int fileId,
    gsi_char url[SAKE_MAX_URL_LENGTH]
);
```

After getting the URL, the file can be download using the GameSpy HTTP SDK, or any other HTTP SDK. After downloading, the headers returned by the server should be checked for the result, to make sure the download was successful.

```
gsi_bool SAKE_CALL sakeGetFileResultFromHeaders
(
    const char *headers,
    SAKEFileResult *result
);
```

You can use `sakeGetFileResultFromHeaders` to do this automatically, or you can check the headers manually for the "Sake-File-Result" header. The value stored in the header is an integer, the possible values of which are enumerated in `SAKEFileResult`. `SAKEFileResult_SUCCESS` means that the file was downloaded successfully, while any other value indicates that there was an error. Note that `sakeGetFileResultFromHeaders` returns `gsi_true` if it was able to find the "Sake-File-Result" header, and `gsi_false` if it was not able to find the header. So the return value does not in itself indicate that the file was downloaded successfully.

Release Process

Games that use Sake are developed using a Sake development backend. This prevents development work from interfering with any live games. When a developer has finished implementing the Sake usage in a game and is ready to start testing the game with the Sake release environment, he should contact GameSpy support at devsupport@gamespy.com.

The developer will need to supply some information on how the game is using Sake, such as what information is being stored in the game's database and how that being accessed and updated. This will allow GameSpy to ensure that the game can be moved to the release environment without any negative impact to that game or to other games. After reviewing this information, GameSpy will move the game's database schema to the release backend. The developer will also need to inform GameSpy if any data should be moved from the development backend to the release backend. The developer will then conduct final testing against the release backend, and GameSpy will monitor the backend to ensure performance.

Appendix I: Default values when creating records with unspecified fields

Sake does not have the concept of NULL data in a record, therefore, if records are created with unspecified fields, the "default" value (listed and set in the Sake Admin site) will be used for these fields upon creating the record.

The only caveat here is for DateAndTime fields. If a DateAndTime field is listed as a defined field when creating a record, but the reported value for this field is NULL, the date will be set to Jan. 01, 1970 by default. If the field is undefined, the default value, `getutcdate()`, is used.

Appendix II: Special Fields used in Sake

In addition to the fields which the developer defines, you can also request values for special intrinsic fields depending on the data being retrieved. Below is a list of extra fields or search tags that can be requested in Sake:

Fileid metadata

Fileids have metadata fields that can be used to obtain extra information about the file. These are formed by following the name of a *fileid* field with a dot and then a string which controls the metadata to be returned. For example, to get the size of a file stored in a field named "video", you would use the field name "video.size". Note that you can request file metadata without requesting the fileid itself.

- **.size** - returns the size of the file in bytes as an Int.
- **.name** - returns the name of the file as an AsciiString, as specified when it was uploaded.
- **.create_time** - returns a DateAndTime value corresponding to when the file was uploaded.
- **.downloads** - returns as an Int the number of times that the file has been downloaded.
- **.profileid** - returns as an Int the profileid of the player that uploaded the file.

Rateable Tables

For Rateable tables, two fields are automatically added to that table - these two fields are special in that they *cannot be updated manually* by a sakeUpdateRecord request (they are updated only by the SDK when rating a record):

- **num_ratings** - stores the number of times that users have given that record a rating.
- **average_rating** - stores the average of all the ratings given to that record.

The maximum range for ratings is 0 through 255 (games can

internally restrict that to a smaller range). Ratings are given as integers, however the average ratings is returned as a floating point number. In addition, players can use the special field tag **my_rating** to obtain their personal rating on a given record. This can also be used when searching for records. So for example, if you wanted to only view records you have rated as > 100 then you would include "my_rating > 100" in the filter string. For records that the player has not yet rated, my_rating is set to -1 by default.

Lastly, there are two special search tags **@rated** and **@unrated** which can be used as SQL filter strings when searching for records to limit the search to rated records (using @rated) or unrated records (using @unrated).

Getting the Row number (or Rank for Leaderboard Queries)

- **row** - The row number (based on given sort criteria). Only usable in a [sakeSearchForRecords](#) request.

To get a player's Rank, it is as simple as adding the special **row** field name to the list of fields to retrieve. This gets the row number based on the sort criteria specified.

Appendix III: Reserved Field names

There are some **reserved** Field Names in Sake that should not be used, otherwise they can cause unforeseen problems. these names are: *file*, *fileid*, *size*, *name*, *create_time*, *downloads*, *profileid*, *ownerid*, *recordid*, *num_ratings*, *average_rating*, *my_rating*.

Appendix IV: Using Sake for ATLAS Leaderboard Queries

SAKE has optimized methods for performing basic leaderboard queries of ATLAS Statistics. The following describes their usage.

1. How to get a player's rank (e.g. so you can display "I'm ranked X...")

To get the player's Rank based on the sort order provided, you will add "**row**" as a special fieldName that returns the player's current row (or in a leaderboard sense, their Rank) to a [sakeSearchForRecords](#) query. Note that this row number is relative to the *mFilter* provided, so this way you can create specific leaderboards (say for example, top 10 players in each gametype). This method works both for getting a single player's rank, or for getting multiple players' ranks.

2. How to get a total record count (e.g. so you can display "... out of X total players")

You can use the [sakeGetRecordCount](#) function which can give you the record count for an entire table (no filter) or for a filtered subset of the table (e.g. how many people with stats recorded for a specific gametype, etc.)

Example Usage (gets the total number of players who have accumulated stats):

```
static SAKEGetRecordCountInput input;
static SAKERequest request;
SAKEStartRequestResult startRequestResult;

input.mTableId = "PlayerStats_v1";
input.mFilter = "";
request = sakeGetRecordCount(sake, &input,;
```

3. How to get a top 10 leaderboard (or a subset of this, such as top 10-20)

you can use a [sakeSearchForRecords](#) query if you want to achieve a top 10 leaderboard for example, and page through it. You will use the parameters: *mFilter*, *mSort*, *mOffset*, *mMaxRecords*. The offset works to page through the leaderboard.

Example Usage - Top 1-10 in CTF Gametype:

```
static SAKEGetRecordCountInput input;
static SAKERequest request;
SAKEStartRequestResult startRequestResult;
static char *fieldNames[] = { "row", "owneri

input.mTableId = "PlayerStats_v1";
input.mFieldNames = fieldNames;
input.mNumFields = (sizeof(fieldNames) / siz

input.mFilter = "GameType = 'CTF'"; // e.g.
input.mSort = "CTF_HighScore desc"; // e.g.
input.mOffset = 0;
input.mMaxRecords = 10;
request = sakeSearchForRecords(sake, &input;
```

Example Usage - Top 11-20 in CTF Gametype:

```
// same as above, just change the offset
input.mOffset = 10;
```

4. How to get my record and those surrounding me

Again you would use a [sakeSearchForRecords](#) query. The parameters this time will include: *mTargetRecordFilter*, *mSurroundingRecordsCount*. The *mTargetRecordFilter* should provide a filter string that will return only a single record, and the *mSurroundingRecordsCount* is how many records above & below that target record to get as well.

Example Usage - My CTF Record & the surrounding 5 CTF players above and below (11 records total possible):

```
static SAKEGetRecordCountInput input;
static SAKERequest request;
SAKEStartRequestResult startRequestResult;
static char *fieldNames[] = { "row", "ownerid"

input.mTableId = "PlayerStats_v1";
input.mFieldNames = fieldNames;
input.mNumFields = (sizeof(fieldNames) / siz

input.mFilter = "GameType = 'CTF'";
input.mSort = "CTF_HighScore desc";
input.mOffset = 0; // c
input.mMaxRecords = 11;
input.mTargetRecordFilter = "ownerid = 81395
input.mSurroundingRecordsCount = 5;

request = sakeSearchForRecords(sake, &input;
```

5. How to get records for a subset of specific profiles (e.g. get stats/ranks for all my friends)

A [sakeSearchForRecords](#) query with parameters now including: *mOwnerIds*, *mNumOwnerIds*. The *mOwnerIds* gives an array of profileids for the subset of records to retrieve, and the *mNumOwnerIds* indicates the number of ids in the array.

Example Usage - My CTF Record & My buddies' CTF records:

```
static SAKEGetRecordCountInput input;
static SAKERequest request;
SAKEStartRequestResult startRequestResult;
static char *fieldNames[] = { "row", "ownerid"
static int ownerIds[5] = { 64880031, 8139534

input.mTableId = "PlayerStats_v1";
```

```
input.mFieldNames = fieldNames;
input.mNumFields = (sizeof(fieldNames) / siz

input.mFilter = "GameType = 'CTF'";
input.mSort = "CTF_HighScore desc";
input.mOffset = 0; // c
input.mMaxRecords = 5;
input.mOwnerIds = ownerIds;
input.mNumOwnerIds = 5;

request = sakeSearchForRecords(sake, &input;
```

Sake SDK Functions

sakeCreateRecord	Creates a new Record in a backend table.
sakeDeleteRecord	Deletes the specified record.
sakeGetFieldByName	This utility function retrieves the specified field from the record, via the name that identifies it.
sakeGetFileDownloadURL	Used to get a download URL for a particular fileid.
sakeGetFileIdFromHeaders	If the file was uploaded successfully, this function obtains the fileid that references the file.
sakeGetFileResultFromHeaders	Checks the headers from the uploaded file to see the result.
sakeGetFileUploadURL	Retrieves the URL which can be used to upload files.
sakeGetMyRecords	Gets all of the records owned by the local player from a table.
sakeGetRandomRecord	Retrieves a random record from the provided search criteria.

<u>sakeGetRecordCount</u>	Gets a count for the number of records in a table based on the given filter criteria.
<u>sakeGetRecordLimit</u>	Checks the maximum number of records that a profile can own for a particular table.
<u>sakeGetSpecificRecords</u>	Gets a list of specific records from a table.
<u>sakeGetStartRequestResult</u>	Called to retrieve the result of a request - normally used to determine the reason for a failed request.
<u>sakeRateRecord</u>	Rates a specified record.
<u>sakeSearchForRecords</u>	Searches a table for records that match certain specified criteria.
<u>sakeSetGame</u>	Authenticates the game to use Sake.
<u>sakeSetProfile</u>	Provides Sake authentication information for the current player.
<u>sakeShutdown</u>	Shuts down the SDK and frees any memory that was allocated for the Sake object.
<u>sakeStartup</u>	Initializes the Sake SDK for use.

[sakeUpdateRecord](#)

Updates the values stored in an existing record.

sakeCreateRecord

Creates a new Record in a backend table.

```
SAKERequest sakeCreateRecord(  
    SAKE sake,  
    SAKECreateRecordInput * input,  
    SAKERequestCallback callback,  
    void * userdata );
```

Routine	Required Header	Distribution
sakeCreateRecord	<sake.h>	SDKZIP

Return Value

Reference to internal object that tracks the request. If this is NULL, then the request has failed to initialize. You can call `sakeGetStartRequestResult` to obtain the reason for the failure.

Parameters

sake

[in] The sake object.

input

[in] Stores the data for the record you wish to create.

callback

[in] The request callback function.

userdata

[in] pointer to user specified data sent to the request callback.

Remarks

If the request completed successfully, then the output object contains the recordid of the newly created record.

Section Reference: [Gamespy Sake SDK](#)

See Also: [SAKECreateRecordInput](#), [SAKECreateRecordOutput](#), [SAKERequestCallback](#)

sakeDeleteRecord

Deletes the specified record.

```
SAKERequest sakeDeleteRecord(  
    SAKE sake,  
    SAKEDeleteRecordInput * input,  
    SAKERequestCallback callback,  
    void * userdata );
```

Routine	Required Header	Distribution
sakeDeleteRecord	<sake.h>	SDKZIP

Return Value

Reference to internal object that tracks the request. If this is NULL, then the request has failed to initialize. You can call `sakeGetStartRequestResult` to obtain the reason for the failure.

Parameters

sake

[in] The sake object.

input

[in] Stores the information for the record you wish to delete.

callback

[in] The request callback function.

userdata

[in] pointer to user specified data sent to the request callback.

Remarks

DeleteRecord does not have an output object, because the backend does not send any response other than the success or failure indicated by the result parameter passed to the callback. When the callback is called, the outputData parameter will always be set to NULL.

Section Reference: [Gamespy Sake SDK](#)

See Also: [SAKEDeleteRecordInput](#), [SAKERequestCallback](#)

sakeGetFieldName

This utility function retrieves the specified field from the record, via the name that identifies it.

```
SAKEField * sakeGetFieldName(  
    const char * name,  
    SAKEField * fields,  
    int numFields );
```

Routine	Required Header	Distribution
sakeGetFieldName	<sake.h>	SDKZIP

Return Value

Pointer to a SAKEField which represents the field that was identified by the given name.

Parameters

name

[in] The name of the field to retrieve.

fields

[in] An array of fields, representing a record.

numFields

[in] The number of fields in the array.

Section Reference: [Gamespy Sake SDK](#)

See Also: [SAKEField](#)

sakeGetFileDownloadURL

Used to get a download URL for a particular fileid.

```
gsi_bool sakeGetFileDownloadURL(  
    SAKE sake,  
    int fileid,  
    gsi_char url );
```

Routine	Required Header	Distribution
sakeGetFileDownloadURL	<sake.h>	SDKZIP

Return Value

gsi_true if download url was retrieved successfully, gsi_false otherwise.

Parameters

sake

[in] The Sake object.

fileId

[in] The fileId returned by the headers of the uploaded file. Call `sakeGetFileIdFromHeaders` to obtain.

url

[out] The download url for the specified file.

Section Reference: [Gamespy Sake SDK](#)

See Also: [sakeGetFileIdFromHeaders](#)

sakeGetFileIdFromHeaders

If the file was uploaded successfully, this function obtains the fileid that references the file.

```
gsi_bool sakeGetFileIdFromHeaders(  
    const char * headers,  
    int * fileid );
```

Routine	Required Header	Distribution
sakeGetFileIdFromHeaders	<sake.h>	SDKZIP

Return Value

gsi_true if able to parse the fileid successfully, gsi_false otherwise.

Parameters

headers

[in] The headers to parse for the fileid. Can obtain these by calling ghttpGetHeaders.

fileId

[ref] reference to the uploaded fileid.

Remarks

To get the fileid from the headers manually, look for the “Sake-File-Id” header. Once obtained, the fileid can now be stored in a fileid field in the database.

*Note that If a file is uploaded, but the fileid is not stored in the database, then the file will be automatically deleted by the backend after approximately 24 hours.

Section Reference: [Gamespy Sake SDK](#)

See Also: [HTTP\ghttpGetHeaders](#)

sakeGetFileResultFromHeaders

Checks the headers from the uploaded file to see the result.

```
gsi_bool sakeGetFileResultFromHeaders(  
    const char * headers,  
    SAKEFileResult * result );
```

Routine	Required Header	Distribution
sakeGetFileResultFromHeaders	<sake.h>	SDKZIP

Return Value

gsi_true if it was able to parse the result successfully, gsi_false otherwise

Parameters

headers

[in] The headers to parse for the fileid. Can obtain these by calling ghttpGetHeaders.

result

[ref] Reference to the result as obtained in the headers.

Remarks

You can also check the headers manually for the “Sake-File-Result” header. The value stored in the header is an integer, the possible values of which are enumerated in SAKEFileResult. SAKEFileResult_SUCCESS means that the file was uploaded successfully, while any other value indicates that there was an error uploading the file.

Section Reference: [Gamespy Sake SDK](#)

See Also: [SAKEFileResult](#)

sakeGetFileUploadURL

Retrieves the URL which can be used to upload files.

```
gsi_bool sakeGetFileUploadURL(  
    SAKE sake,  
    gsi_char url );
```

Routine	Required Header	Distribution
sakeGetFileUploadURL	<sake.h>	SDKZIP

Return Value

gsi_true if the upload URL was obtained, gsi_false otherwise

Parameters

sake

[in] The SAKE object.

url

[out] The URL where the file can be uploaded.

Section Reference: [Gamespy Sake SDK](#)

sakeGetMyRecords

Gets all of the records owned by the local player from a table.

```
SAKERequest sakeGetMyRecords(  
    SAKE sake,  
    SAKEGetMyRecordsInput * input,  
    SAKERequestCallback callback,  
    void * userData );
```

Routine	Required Header	Distribution
sakeGetMyRecords	<sake.h>	SDKZIP

Return Value

Reference to internal object that tracks the request. If this is NULL, then the request has failed to initialize. You can call `sakeGetStartRequestResult` to obtain the reason for the failure.

Parameters

sake

[in] The Sake object.

input

[in] Stores info about the records you wish to retrieve.

callback

[in] The request callback function.

userData

[in] pointer to user specified data sent to the request callback.

Remarks

If the request completed successfully, then the output object contains all of the records which the local player owns in the table. See definitions of the Input & Output structs for more information about how to limit what is retrieved in the request and certain metadata fields that can be retrieved.

Section Reference: [Gamespy Sake SDK](#)

See Also: [SAKEGetMyRecordsInput](#), [SAKEGetMyRecordsOutput](#), [SAKERestRequestCallback](#)

sakeGetRandomRecord

Retrieves a random record from the provided search criteria.

```
SAKERequest sakeGetRandomRecord(  
    SAKE sake,  
    SAKEGetRandomRecordInput * input,  
    SAKERequestCallback callback,  
    void * userData );
```

Routine	Required Header	Distribution
sakeGetRandomRecord	<sake.h>	SDKZIP

Return Value

Reference to internal object that tracks the request. If this is NULL, then the request has failed to initialize. You can call `sakeGetStartRequestResult` to obtain the reason for the failure.

Parameters

sake

[in] The sake object.

input

[in] Stores the info for the random record search.

callback

[in] The request callback function.

userData

[in] pointer to user specified data sent to the request callback.

Remarks

The output will always be a single record (unless no records pass the filter, in which case the output will contain NULL data for the returned record). Note that this function works best in a table in which records are not deleted or are deleted in order of oldest first (in other words, tables where records are contiguous).

Section Reference: [Gamespy Sake SDK](#)

See Also: [SAKEGetRandomRecordInput](#),
[SAKEGetRandomRecordOutput](#), [SAKERequestCallback](#)

sakeGetRecordCount

Gets a count for the number of records in a table based on the given filter criteria.

```
SAKERequest sakeGetRecordCount(  
    SAKE sake,  
    SAKEGetRecordCountInput * input,  
    SAKERequestCallback callback,  
    void * userData );
```

Routine	Required Header	Distribution
sakeGetRecordCount	<sake.h>	SDKZIP

Return Value

Reference to internal object that tracks the request. If this is NULL, then the request has failed to initialize. You can call `sakeGetStartRequestResult` to obtain the reason for the failure.

Parameters

sake

[in] The sake object.

input

[in] Stores info on the table whose count you wish to check.

callback

[in] The request callback function.

userData

[in] pointer to user specified data sent to the request callback.

Remarks

If the request completed successfully, then the Output object contains info about the count for the specified table.

Section Reference: [Gamespy Sake SDK](#)

See Also: [SAKEGetRecordCountInput](#), [SAKEGetRecordCountOutput](#), [SAKERestRequestCallback](#)

sakeGetRecordLimit

Checks the maximum number of records that a profile can own for a particular table.

```
SAKERequest sakeGetRecordLimit(  
    SAKE sake,  
    SAKEGetRecordLimitInput * input,  
    SAKERequestCallback callback,  
    void * userData );
```

Routine	Required Header	Distribution
sakeGetRecordLimit	<sake.h>	SDKZIP

Return Value

Reference to internal object that tracks the request. If this is NULL, then the request has failed to initialize. You can call `sakeGetStartRequestResult` to obtain the reason for the failure.

Parameters

sake

[in] The sake object.

input

[in] Stores info on the table whose record limit you wish to check.

callback

[in] The request callback function.

userData

[in] pointer to user specified data sent to the request callback.

Remarks

If the request completed successfully, then the Output object contains info about the record limit for the specified table.

Section Reference: [Gamespy Sake SDK](#)

See Also: [SAKEGetRecordLimitInput](#), [SAKEGetRecordLimitOutput](#), [SAKERestRequestCallback](#)

sakeGetSpecificRecords

Gets a list of specific records from a table.

```
SAKERequest sakeGetSpecificRecords(  
    SAKE sake,  
    SAKEGetSpecificRecordsInput * input,  
    SAKERequestCallback callback,  
    void * userData );
```

Routine	Required Header	Distribution
sakeGetSpecificRecords	<sake.h>	SDKZIP

Return Value

Reference to internal object that tracks the request. If this is NULL, then the request has failed to initialize. You can call `sakeGetStartRequestResult` to obtain the reason for the failure.

Parameters

sake

[in] The sake object.

input

[in] Stores the info about the specific records you wish to retrieve.

callback

[in] The request callback function.

userData

[in] pointer to user specified data sent to the request callback.

Remarks

If the request completed successfully, then the output object contains all of the records which were specified in the request. See definitions of the Input & Output structs for more information about how to limit what is retrieved in the request and certain metadata fields that can be retrieved.

Section Reference: [Gamespy Sake SDK](#)

See Also: [SAKEGetSpecificRecordsInput](#),
[SAKEGetSpecificRecordsOutput](#), [SAKERequestCallback](#)

sakeGetStartRequestResult

Called to retrieve the result of a request - normally used to determine the reason for a failed request.

```
SAKEStartRequestResult sakeGetStartRequestResult(  
SAKE sake );
```

Routine	Required Header	Distribution
sakeGetStartRequestResult	<sake.h>	SDKZIP

Return Value

Enum value used to indicate the specific result of the request.

Parameters

sake

[in] The sake object

Remarks

This function will always return the most recent request that was attempted, so it must be called immediately after a failure to get the reason for that failure.

Section Reference: [Gamespy Sake SDK](#)

See Also: [SAKEStartRequestResult](#)

sakeRateRecord

Rates a specified record.

```
SAKERequest sakeRateRecord(  
    SAKE sake,  
    SAKERateRecordInput * input,  
    SAKERequestCallback callback,  
    void * userdata );
```

Routine	Required Header	Distribution
sakeRateRecord	<sake.h>	SDKZIP

Return Value

Reference to internal object that tracks the request. If this is NULL, then the request has failed to initialize. You can call `sakeGetStartRequestResult` to obtain the reason for the failure.

Parameters

sake

[in] The sake object.

input

[in] Stores the information for the record you wish to rate.

callback

[in] The request callback function.

userdata

[in] pointer to user specified data sent to the request callback.

Remarks

The range of ratings which Sake supports is 0 to 255. However a game can restrict itself to a subset of that range if it wishes. For example, a game may want to use a rating of 1 to 5 (a star rating), or it may want to use a range of 0 to 100. Sake allows users to rate records which they own, however no profile can rate a single record more than once.

Sake stores on the backend each individual rating that has been given, which allows it to compute accurate averages and prevent repeat ratings. The field name "my_rating" can be used to obtain the current profile's rating for a given record or when searching for records. By default, my_rating = -1 for records that have not yet been rated. When browsing for records, use the special search tags @rated or @unrated in the filter string to limit the searches to either rated or unrated records.

Section Reference: [Gamespy Sake SDK](#)

See Also: [SAKERateRecordInput](#), [SAKERRequestCallback](#)

sakeSearchForRecords

Searches a table for records that match certain specified criteria.

```
SAKERequest sakeSearchForRecords(  
    SAKE sake,  
    SAKESearchForRecordsInput * input,  
    SAKERequestCallback callback,  
    void * userData );
```

Routine	Required Header	Distribution
sakeSearchForRecords	<sake.h>	SDKZIP

Return Value

Reference to internal object that tracks the request. If this is NULL, then the request has failed to initialize. You can call `sakeGetStartRequestResult` to obtain the reason for the failure.

Parameters

sake

[in] The sake object.

input

[in] Stores the info about the records you wish to search for.

callback

[in] The request callback function.

userData

[in] pointer to user specified data sent to the request callback.

Remarks

If the request completed successfully, then the output object contains contains records founds by the search. See definitions of the Input & Output structs for more information about how to limit what is retrieved in the request and certain metadata fields that can be retrieved.

Section Reference: [Gamespy Sake SDK](#)

See Also: [SAKESearchForRecordsInput](#),
[SAKESearchForRecordsOutput](#), [SAKERequestCallback](#)

sakeSetGame

Authenticates the game to use Sake.

```
void sakeSetGame(  
    SAKE sake,  
    const char * gameName,  
    int gameId,  
    const char * secretKey );
```

Routine	Required Header	Distribution
sakeSetGame	<sake.h>	SDKZIP

Parameters

sake

[in] The Sake object.

gameName

[in] Your title's gamename, assigned by GameSpy.

gameId

[in] Your title's gameid, assigned by GameSpy.

secretKey

[in] Your title's secret key , assigned by GameSpy.

Remarks

The function provides no indication of whether or not the gamename and gameid are correct. The backend will simply check them and use the information to figure out which game's database is being used.

The game should also call `sakeSetProfile` to authenticate the current player before continuing with any other Sake usage.

Section Reference: [Gamespy Sake SDK](#)

See Also: [sakeSetProfile](#)

sakeSetProfile

Provides Sake authentication information for the current player.

```
void sakeSetProfile(  
    SAKE sake,  
    int profileId,  
    const char * loginTicket );
```

Routine	Required Header	Distribution
sakeSetProfile	<sake.h>	SDKZIP

Parameters

sake

[in] The sake object.

profileId

[in] Current player's profile ID.

loginTicket

[in] Current player's login ticket, which allows the backend to verify the player is correctly identifying himself.

Remarks

The profile ID and login ticket are both obtained from the GameSpy Presence and Messaging SDK (GP).

The profile ID can be obtained in the callback that is called as a result of logging into GP. A GPConnectResponseArg struct is passed to the callback, and the struct has a member variable “profile” that stores the player’s profile ID. While the player is logged on, the game should call the GP function gpGetLoginTicket.

As with sakeSetGame, **sakeSetProfile** provides no indication of whether or not the information provided is correct. The backend checks these values and uses them to authenticate the player and, for certain requests, identify which player’s data is being access or updated.

Section Reference: [Gamespy Sake SDK](#)

See Also: [Presence\GPConnectResponseArg](#),
[Presence\gpGetLoginTicket](#)

sakeShutdown

Shuts down the SDK and frees any memory that was allocated for the Sake object.

```
void sakeShutdown(  
    SAKE sake );
```

Routine	Required Header	Distribution
sakeShutdown	<sake.h>	SDKZIP

Parameters

sake

[in] The sake object.

Remarks

After this function returns, the reference to the Sake object is no longer valid and should not be used. The game should also shutdown the GameSpy Core object by calling gsCoreShutdown. Sample code for this is available in the Sake test app.

Section Reference: [Gamespy Sake SDK](#)

sakeStartup

Initializes the Sake SDK for use.

```
SAKEStartupResult sakeStartup(  
    SAKE * sakePtr );
```

Routine	Required Header	Distribution
sakeStartup	<sake.h>	SDKZIP

Return Value

An enumeration of possible results. If the result is `SAKEStartupResult_SUCCESS`, then the startup has succeeded. Any other value indicates a failure, and the game should not continue calling other Sake functions.

Parameters

sakePtr

[in] Pointer to a Sake object, which is initialized by startup. This will be used in nearly all subsequent Sake calls.

Remarks

Before using Sake, the GameSpy Availability Check must have been performed and indicated that the game's backend is available, and the Core object must have been initialized by calling `gsCoreInitialize`. Sample code for this is available in the Sake test app.

The SAKE object initialized by this startup call is valid until the game shutdowns the Sake SDK with `sakeShutdown`.

Section Reference: [Gamespy Sake SDK](#)

See Also: [SAKEStartupResult](#)

sakeUpdateRecord

Updates the values stored in an existing record.

```
SAKERequest sakeUpdateRecord(  
    SAKE sake,  
    SAKEUpdateRecordInput * input,  
    SAKERequestCallback callback,  
    void * userdata );
```

Routine	Required Header	Distribution
sakeUpdateRecord	<sake.h>	SDKZIP

Return Value

Reference to internal object that tracks the request. If this is NULL, then the request has failed to initialize. You can call `sakeGetStartRequestResult` to obtain the reason for the failure.

Parameters

sake

[in] The sake object.

input

[in] Stores the updated data for the record.

callback

[in] The request callback function.

userdata

[in] pointer to user specified data sent to the request callback.

Remarks

UpdateRecord does not have an output object, because the backend does not send any response other than the success or failure indicated by the result parameter passed to the callback. When the callback is called, the outputData parameter will always be set to NULL.

Section Reference: [Gamespy Sake SDK](#)

See Also: [SAKEUpdateRecordInput](#), [SAKERequestCallback](#)

Sake SDK Callbacks

[SAKERequestCallback](#)

Callback called when a request completes.

SAKERequestCallback

Callback called when a request completes.

```
typedef void (*SAKERequestCallback)(  
    SAKE sake,  
    SAKERequest request,  
    SAKERequestResult result,  
    void *inputData,  
    void *outputData,  
    void *userData );
```

Routine	Required Header	Distribution
SAKERequestCallback	<sake.h>	SDKZIP

Parameters

sake

[out] The sake object.

request

[out] State of the sake request.

result

[out] The result of the request; `SAKERequestResult_SUCCESS` means success, any other value indicates failure.

inputData

[ref] Pointer to the input object which was passed into the request.

outputData

[ref] Pointer to an output object for this request, which should be cast to the appropriate type based on the request made.

userData

[ref] Pointer to the `userData` passed into the request function.

Remarks

Not all request types have output objects. If a request type does not have an output object, then `outputData` will be always be NULL when the callback is called.

Section Reference: [Gamespy Sake SDK](#)

See Also: [SAKERequestResult](#)

Sake SDK Structures

SAKEBinaryData	Data struct used to store arbitrary binary data in a Sake field.
SAKECreateRecordInput	Input object passed to sakeCreateRecord.
SAKECreateRecordOutput	Returned output object that specifies the recordid for the newly created record.
SAKEDeleteRecordInput	Input object passed to sakeDeleteRecord.
SAKEField	object used to represent the field of a record.
SAKEGetMyRecordsInput	Input object passed to sakeGetMyRecords.
SAKEGetMyRecordsOutput	Returned output object that specifies all of the records which the local player owns in the table.
SAKEGetRandomRecordInput	Input object passed to sakeGetRandomRecord.
SAKEGetRandomRecordOutput	Returned output object that contains a random record.

SAKEGetRecordCountInput	Input object passed to sakeGetRecordCount
SAKEGetRecordCountOutput	Returned record count based on the specified table and search filter used.
SAKEGetRecordLimitInput	Input object passed to sakeGetRecordLimit.
SAKEGetRecordLimitOutput	Returned output object that specifies the maximum number of records that a profile can own in the table.
SAKEGetSpecificRecordsInput	Input object passed to sakeGetSpecificRecords.
SAKEGetSpecificRecordsOutput	Returned output object that contains all of the records which were specified in the request.
SAKERateRecordInput	Input object passed to sakeRateRecord.
SAKERateRecordOutput	Returned output object that lists the new number of ratings and the new average rating for the specified record.
SAKESearchForRecordsInput	Input object passed to sakeSearchForRecords.

[SAKESearchForRecordsOutput](#) Returned output object that contains the records founds by the search.

[SAKEUpdateRecordInput](#) Input object passed to sakeUpdateRecord.

SAKEBinaryData

Data struct used to store arbitrary binary data in a Sake field.

```
typedef struct  
{  
    gsi_u8 * mValue;  
    int mLength;  
} SAKEBinaryData;
```

Members

mValue

pointer to the data.

mLength

the number of bytes of data.

Remarks

mValue may be NULL if mLength is 0.

Section Reference: [Gamespy Sake SDK](#)

See Also: [SAKEFieldType](#)

SAKECreateRecordInput

Input object passed to sakeCreateRecord.

```
typedef struct  
{  
    char * mTableId;  
    SAKEField * mFields;  
    int mNumFields;  
} SAKECreateRecordInput;
```

Members

mTableId

Points to the tableid of the table in which the record will be created.

mFields

Points to an array of fields which has the initial values for the new record's fields.

mNumFields

Stores the number of fields in the mFields array.

Remarks

Any fields which are not initialized will be set to their default value. If mNumFields is 0, indicating that no initial values will be set, then mFields can be NULL.

Section Reference: [Gamespy Sake SDK](#)

See Also: [sakeCreateRecord](#)

SAKECreateRecordOutput

Returned output object that specifies the recordid for the newly created record.

```
typedef struct  
{  
    int mRecordId;  
} SAKECreateRecordOutput;
```

Members

mRecordId

The recordid for the newly created record.

Section Reference: [Gamespy Sake SDK](#)

See Also: [sakeCreateRecord](#), [SAKERequestCallback](#)

SAKEDeleteRecordInput

Input object passed to sakeDeleteRecord.

```
typedef struct  
{  
    char * mTableId;  
    int mRecordId;  
} SAKEDeleteRecordInput;
```

Members

mTableId

Points to the tableid of the table in which the record to be deleted exists.

mRecordId

Identifies the record to be deleted.

Remarks

DeleteRecord does not have an output object, because the backend does not send any response other than the success or failure indicated by the result parameter passed to the callback. When the callback is called, the outputData parameter will always be set to NULL.

Section Reference: [Gamespy Sake SDK](#)

See Also: [sakeDeleteRecord](#)

SAKEField

object used to represent the field of a record.

```
typedef struct  
{  
    char * mName;  
    SAKEFieldType mType;  
    SAKEValue mValue;  
} SAKEField;
```

Members

mName

the name used to identify the field.

mType

The type of data stored in the field.

mValue

The value that will be stored in the field.

Section Reference: [Gamespy Sake SDK](#)

See Also: [SAKEFieldType](#), [SAKEBinaryData](#)

SAKEGetMyRecordsInput

Input object passed to sakeGetMyRecords.

```
typedef struct  
{  
    char * mTableId;  
    char ** mFieldNames;  
    int mNumFields;  
} SAKEGetMyRecordsInput;
```

Members

mTableId

Points to the tableid of the table from which to return records.

mFieldNames

points to an array of strings, each of which contains the name of a field for which to return values.

mNumFields

stores the number of strings in the mFieldNames array. This list controls the values which will be returned as part of the response. The array can contain just one field name, the names of all the fields in the table, or any subset of the field names.

Remarks

In addition to the fields which the developer defines, you can also request values for the “recordid” field, “ownerid” field (if the table has an owner type of profile), and “num_ratings” and “average_rating” fields (if the table has its ratings option set to true).

See Appendix II in the Overview for more information on special fields used in Sake.

Section Reference: [Gamespy Sake SDK](#)

See Also: [sakeGetMyRecords](#)

SAKEGetMyRecordsOutput

Returned output object that specifies all of the records which the local player owns in the table.

```
typedef struct  
{  
    int mNumRecords;  
    SAKEField ** mRecords;  
} SAKEGetMyRecordsOutput;
```

Members

mNumRecords

The number of records found.

mRecords

Points an array of records, each of which is represented as an array of fields.

Section Reference: [Gamespy Sake SDK](#)

See Also: [sakeGetMyRecords](#), [SAKERestRequestCallback](#), [SAKEField](#)

SAKEGetRandomRecordInput

Input object passed to sakeGetRandomRecord.

```
typedef struct  
{  
    char * mTableId;  
    char ** mFieldNames;  
    int mNumFields;  
    char * mFilter;  
} SAKEGetRandomRecordInput;
```

Members

mTableId

Points to the tableid of the table to be searched.

mFieldNames

Points to an array of strings, each of which contains the name of a field for which to return values. This list controls the values which will be returned as part of the response. The array can contain just one field name, the names of all the fields in the table, or any subset of the field names.

mNumFields

Stores the number of strings in the *mFieldNames* array.

mFilter

SQL-like filter string which is used to filter which records are to be looked at when choosing a random record. Note that if the search criteria is too specific and no records are found, then the output will return no random record. Note that a field can be used in the filter string even if it is not listed in the *mFieldNames* array, and that file metadata fields can be used in a filter string.

Remarks

In addition to the fields which the developer defines, you can also request values for the “recordid” field, “ownerid” field (if the table has an owner type of profile), and “num_ratings” and “average_rating” fields (if the table has its ratings option set to true).

See Appendix II in the Overview for more information on special fields used in Sake.

Section Reference: [Gamespy Sake SDK](#)

See Also: [sakeGetRandomRecord](#)

SAKEGetRandomRecordOutput

Returned output object that contains a random record.

```
typedef struct  
{  
    SAKEField * mRecord;  
} SAKEGetRandomRecordOutput;
```

Members

mRecord

An array of fields representing the random record.

Remarks

If no record was found due to constrained search criteria, the returned record will be set to NULL.

Section Reference: [Gamespy Sake SDK](#)

SAKEGetRecordCountInput

Input object passed to sakeGetRecordCount.

```
typedef struct  
{  
    char * mTableId;  
    char * mFilter;  
    gsi_bool mCacheFlag;  
} SAKEGetRecordCountInput;
```

Members

mTableId

Points to the tableid of the table to be searched.

mFilter

SQL-like filter string which is used to filter which records are to be looked at when getting the record count.

mCacheFlag

Enables caching if set to `gsi_true`. Defaults to no caching if none is specified.

Section Reference: [Gamespy Sake SDK](#)

See Also: [sakeGetRecordCount](#)

SAKEGetRecordCountOutput

Returned record count based on the specified table and search filter used.

```
typedef struct  
{  
    int mCount;  
} SAKEGetRecordCountOutput;
```

Members

mCount

Contains the value of the record count. If no records exist or the search criteria was too specific so that no records were found, this value will be 0.

Section Reference: [Gamespy Sake SDK](#)

See Also: [sakeGetRecordCount](#), [SAKERequestCallback](#)

SAKEGetRecordLimitInput

Input object passed to sakeGetRecordLimit.

```
typedef struct  
{  
    char * mTableId;  
} SAKEGetRecordLimitInput;
```

Members

mTableId

Points to the tableid of the table for which to check the limit.

Section Reference: [Gamespy Sake SDK](#)

See Also: [sakeGetRecordLimit](#)

SAKEGetRecordLimitOutput

Returned output object that specifies the maximum number of records that a profile can own in the table.

```
typedef struct  
{  
    int mLimitPerOwner;  
    int mNumOwned;  
} SAKEGetRecordLimitOutput;
```

Members

mLimitPerOwner

Contains the maximum number of records that a profile can own in the table; corresponds to the limit per owner option that can be set using the Sake Administration website.

mNumOwned

Contains the number of records that the local profile currently owns in the table.

Section Reference: [Gamespy Sake SDK](#)

See Also: [sakeGetRecordLimit](#), [SAKERRequestCallback](#)

SAKEGetSpecificRecordsInput

Input object passed to sakeGetSpecificRecords.

```
typedef struct  
{  
    char * mTableId;  
    int * mRecordIds;  
    int mNumRecordIds;  
    char ** mFieldNames;  
    int mNumFields;  
} SAKEGetSpecificRecordsInput;
```

Members

mTableId

Points to the tableid of the table from which to get the records.

mRecordIds

An array of recordids, each one identifying a record which is to be returned.

mNumRecordIds

The number of recordids in the mRecordIds array.

mFieldNames

Points to an array of strings, each of which contains the name of a field for which to return values.

mNumFields

stores the number of strings in the mFieldNames array. This list controls the values which will be returned as part of the response. The array can contain just one field name, the names of all the fields in the table, or any subset of the field names.

Remarks

In addition to the fields which the developer defines, you can also request values for the “recordid” field, “ownerid” field (if the table has an owner type of profile), and “num_ratings” and “average_rating” fields (if the table has its ratings option set to true).

See Appendix II in the Overview for more information on special fields used in Sake.

Section Reference: [Gamespy Sake SDK](#)

See Also: [sakeGetSpecificRecords](#)

SAKEGetSpecificRecordsOutput

Returned output object that contains all of the records which were specified in the request.

```
typedef struct  
{  
    int mNumRecords;  
    SAKEField ** mRecords;  
} SAKEGetSpecificRecordsOutput;
```

Members

mNumRecords

The number of records found.

mRecords

Points an array of records, each of which is represented as an array of fields.

Section Reference: [Gamespy Sake SDK](#)

See Also: [sakeGetSpecificRecords](#), [SAKERequestCallback](#), [SAKEField](#)

SAKERateRecordInput

Input object passed to sakeRateRecord.

```
typedef struct  
{  
    char * mTableId;  
    int mRecordId;  
    gsi_u8 mRating;  
} SAKERateRecordInput;
```

Members

mTableId

Points to the tableid of the table in which the record to be rated exists.

mRecordId

The recordid of the record to rate.

mRating

The rating the user wants to give the record.

Remarks

The range of ratings which Sake supports is 0 to 255. However a game can restrict itself to a subset of that range if it wishes. For example, a game may want to use a rating of 1 to 5 (a star rating), or it may want to use a range of 0 to 100. Sake allows users to rate records which they own, however no profile can rate a single record more than once. The special field "my_rating" keeps track of what the current profile's rating is for a given record; if the record has not yet been rated, my_rating = -1 by default.

Section Reference: [Gamespy Sake SDK](#)

See Also: [sakeRateRecord](#)

SAKERateRecordOutput

Returned output object that lists the new number of ratings and the new average rating for the specified record.

```
typedef struct  
{  
    int mNumRatings;  
    float mAverageRating;  
} SAKERateRecordOutput;
```

Members

mNumRatings

The number of ratings associated with this record.

mAverageRating

The average rating of this record.

Section Reference: [Gamespy Sake SDK](#)

See Also: [sakeRateRecord](#), [SAKERRequestCallback](#)

SAKESearchForRecordsInput

Input object passed to sakeSearchForRecords.

```
typedef struct  
{  
    char * mTableId;  
    char ** mFieldNames;  
    int mNumFields;  
    char * mFilter;  
    char * mSort;  
    int mOffset;  
    int mMaxRecords;  
    char * mTargetRecordFilter;  
    int mSurroundingRecordsCount;  
    int * mOwnerIds;  
    int mNumOwnerIds;  
    gsi_bool mCacheFlag;  
} SAKESearchForRecordsInput;
```

Members

mTableId

Points to the tableid of the table to be searched.

mFieldNames

Points to an array of strings, each of which contains the name of a field for which to return values. This list controls the values which will be returned as part of the response. The array can contain just one field name, the names of all the fields in the table, or any subset of the field names.

mNumFields

Stores the number of strings in the *mFieldNames* array.

mFilter

SQL-like filter string which is used to search for records based on the values in their fields. For example, to find everyone who has a score of more than 50 use "score > 50", or to find everyone who has a name that starts with an A use "name like 'A%'". Note that a field can be used in the filter string even if it is not listed in the *mFieldNames* array, and that file metadata fields can be used in a filter string.

mSort

SQL-like sort string which is used to sort the records which are found by the search. To sort the results on a particular field, just pass in the name of that field, and the results will be sorted from lowest to highest based on that field. To make the sort descending instead of ascending add " desc" after the name of the field. Note that a field can be used in the sort string even if it is not listed in the *mFieldNames* array, and that file metadata fields can be used in a sort.

mOffset

If not set to 0, then the backend will return records starting from the given offset into the result set.

mMaxRecords

Used to specify the maximum number of records to return for a particular search.

mTargetRecordFilter

Used to specify a single record to return - when done in conjunction with `mSurroundingRecordsCount`, this will return the "target" record plus the surrounding records above and below this target record. Can also be used to specify a "set" of target records to return, but when used in this context the surrounding records count does not apply.

mSurroundingRecordsCount

Used in conjunction with `mTargetRecordFilter` - specifies the number of records to return above and below the target record. (e.g. if = 5, you will receive a maximum of 11 possible records, the target record + 5 above and 5 below).

mOwnerIds

Specifies an array of ownerIds (profileid of record owner) to return from the search.

mNumOwnerIds

Specifies the number of ids contained in the `mOwnerIds` array.

mCacheFlag

Enables caching if set to `gsi_true`. Defaults to no caching if none is specified.

Remarks

In addition to the fields which the developer defines, you can also request values for the “recordid” field, “ownerid” field (if the table has an owner type of profile), and “num_ratings” and “average_rating” fields (if the table has its ratings option set to true).

See Appendix II in the Overview for more information on special fields used in Sake.

Section Reference: [Gamespy Sake SDK](#)

See Also: [sakeSearchForRecords](#)

SAKESearchForRecordsOutput

Returned output object that contains the records founds by the search.

```
typedef struct  
{  
    int mNumRecords;  
    SAKEField ** mRecords;  
} SAKESearchForRecordsOutput;
```

Members

mNumRecords

The number of records found.

mRecords

Points an array of records, each of which is represented as an array of fields.

Section Reference: [Gamespy Sake SDK](#)

See Also: [sakeSearchForRecords](#), [SAKERequestCallback](#), [SAKEField](#)

SAKEUpdateRecordInput

Input object passed to sakeUpdateRecord.

```
typedef struct  
{  
    char * mTableId;  
    int mRecordId;  
    SAKEField * mFields;  
    int mNumFields;  
} SAKEUpdateRecordInput;
```

Members

mTableId

Points to the tableid of the table in which the record to be updated exists.

mRecordId

Identifies the record to be updated.

mFields

Points to an array of fields which has the new values for the record's fields.

mNumFields

Stores the number of fields in the mFields array.

Remarks

Unlike with a CreateRecord request, mNumFields cannot be 0; at least one field must be updated.

UpdateRecord does not have an output object, because the backend does not send any response other than the success or failure indicated by the result parameter passed to the callback. When the callback is called, the outputData parameter will always be set to NULL.

Section Reference: [Gamespy Sake SDK](#)

See Also: [sakeUpdateRecord](#)

Sake SDK Enumerations

SAKEFieldType	Indicates the type of data stored in a field.
SAKEFileResult	Used to determine the status of a file uploaded to Sake.
SAKERequestResult	The result of Sake calls used to modify or read from records (returned to the <code>SAKERequestCallback</code>).
SAKEStartRequestResult	The status result of the most recent request.
SAKEStartupResult	value returned from the call to <code>sakeStartup</code> .

SAKEFieldType

Indicates the type of data stored in a field.

typedef enum

```
{  
    SAKEFieldType_BYTE,  
    SAKEFieldType_SHORT,  
    SAKEFieldType_INT,  
    SAKEFieldType_FLOAT,  
    SAKEFieldType_ASCII_STRING,  
    SAKEFieldType_UNICODE_STRING,  
    SAKEFieldType_BOOLEAN,  
    SAKEFieldType_DATE_AND_TIME,  
    SAKEFieldType_BINARY_DATA,  
    SAKEFieldType_NUM_FIELD_TYPES  
} SAKEFieldType;
```

Remarks

The value for a field is stored in a SAKEValue union.

Section Reference: [Gamespy Sake SDK](#)

See Also: [SAKEField](#), [SAKEBinaryData](#)

SAKEFileResult

Used to determine the status of a file uploaded to Sake.

typedef enum

{

SAKEFileResult_SUCCESS,

SAKEFileResult_BAD_HTTP_METHOD,

SAKEFileResult_BAD_FILE_COUNT,

SAKEFileResult_MISSING_PARAMETER,

SAKEFileResult_FILE_NOT_FOUND,

SAKEFileResult_FILE_TOO_LARGE,

SAKEFileResult_SERVER_ERROR,

SAKEFileResult_UNKNOWN_ERROR

} SAKEFileResult;

Constants

SAKEFileResult_SUCCESS

Upload succeeded.

SAKEFileResult_BAD_HTTP_METHOD

Incorrect ghttp call used to upload file.

SAKEFileResult_BAD_FILE_COUNT

Number of files uploaded is incorrect.

SAKEFileResult_MISSING_PARAMETER

Missing parameter in the ghttp upload call.

SAKEFileResult_FILE_NOT_FOUND

No file was found.

SAKEFileResult_FILE_TOO_LARGE

File uploaded larger than the specified size.

SAKEFileResult_SERVER_ERROR

Unknown error occurred on the server when processing this request.

SAKEFileResult_UNKNOWN_ERROR

Error is unknown - used if none of the above.

Section Reference: [Gamespy Sake SDK](#)

See Also: [sakeGetFileResultFromHeaders](#)

SAKERequestResult

The result of Sake calls used to modify or read from records (returned to the SAKERequestCallback).

typedef enum

```
{  
    SAKERequestResult_SUCCESS,  
    SAKERequestResult_SECRET_KEY_INVALID,  
    SAKERequestResult_SERVICE_DISABLED,  
    SAKERequestResult_CONNECTION_TIMEOUT,  
    SAKERequestResult_CONNECTION_ERROR,  
    SAKERequestResult_MALFORMED_RESPONSE,  
    SAKERequestResult_OUT_OF_MEMORY,  
    SAKERequestResult_DATABASE_UNAVAILABLE,  
    SAKERequestResult_LOGIN_TICKET_INVALID,  
    SAKERequestResult_LOGIN_TICKET_EXPIRED,  
    SAKERequestResult_TABLE_NOT_FOUND,  
    SAKERequestResult_RECORD_NOT_FOUND,  
    SAKERequestResult_FIELD_NOT_FOUND,  
    SAKERequestResult_FIELD_TYPE_INVALID,  
    SAKERequestResult_NO_PERMISSION,  
    SAKERequestResult_RECORD_LIMIT_REACHED,  
    SAKERequestResult_ALREADY_RATED,  
    SAKERequestResult_NOT_RATEABLE,  
    SAKERequestResult_NOT_OWNED,  
    SAKERequestResult_FILTER_INVALID,  
    SAKERequestResult_SORT_INVALID,  
    SAKERequestResult_UNKNOWN_ERROR  
} SAKERequestResult;
```

Section Reference: [Gamespy Sake SDK](#)

See Also: [SAKERequestCallback](#)

SAKEStartRequestResult

The status result of the most recent request.

typedef enum

```
{  
    SAKEStartRequestResult_SUCCESS,  
    SAKEStartRequestResult_NOT_AUTHENTICATED,  
    SAKEStartRequestResult_OUT_OF_MEMORY,  
    SAKEStartRequestResult_BAD_INPUT,  
    SAKEStartRequestResult_BAD_TABLEID,  
    SAKEStartRequestResult_BAD_FIELDS,  
    SAKEStartRequestResult_BAD_NUM_FIELDS,  
    SAKEStartRequestResult_BAD_FIELD_NAME,  
    SAKEStartRequestResult_BAD_FIELD_TYPE,  
    SAKEStartRequestResult_BAD_FIELD_VALUE,  
    SAKEStartRequestResult_BAD_OFFSET,  
    SAKEStartRequestResult_BAD_MAX,  
    SAKEStartRequestResult_BAD_RECORDIDS,  
    SAKEStartRequestResult_BAD_NUM_RECORDIDS,  
    SAKEStartRequestResult_UNKNOWN_ERROR  
} SAKEStartRequestResult;
```

Section Reference: [Gamespy Sake SDK](#)

See Also: [sakeGetStartRequestResult](#)

SAKEStartupResult

value returned from the call to sakeStartup.

typedef enum

{

SAKEStartupResult_SUCCESS,

SAKEStartupResult_NOT_AVAILABLE,

SAKEStartupResult_CORE_SHUTDOWN,

SAKEStartupResult_OUT_OF_MEMORY

} SAKEStartupResult;

Constants

SAKEStartupResult_SUCCESS

Startup succeeded.

SAKEStartupResult_NOT_AVAILABLE

The Sake backend is unavailable.

SAKEStartupResult_CORE_SHUTDOWN

Error in the gsCore.

SAKEStartupResult_OUT_OF_MEMORY

Not enough memory to initialize Sake.

Section Reference: [Gamespy Sake SDK](#)

See Also: [sakeStartup](#)

Server Browsing SDK

Overview

The GameSpy Server Browsing SDK is a portable LAN and Internet server browser engine. It allows developers to quickly and easily add a list-based matchmaking interface to the game, with powerful features such as server-side filtering, sorting, country-filtering, and ping (latency) measurement.

The concept of Server Browsing was popularized by our original GameSpy3D product, and is used as a matchmaking paradigm by GameSpy Arcade and many other online services today. The system functions as follows:

1. A game server (or person hosting a game) starts, and reports its presence to our Master Server. This server reporting is done using the Query & Reporting SDK.
2. Our Master Server aggregates a list of all available game servers, as well as all of the data known about the servers.
3. Game Clients query the Master Server for a list of available game servers. This query can contain a filter to narrow down the list of servers returned.
4. Once the list is obtained, the Game Client queries each server to obtain the latest information about the game (the name of the game, map being played, number of players, or any other relevant information). It also measures latency to the server at this time, since latency can be an important factor in the quality of the game play experience.
5. This collection of servers is then displayed to the user, and they are able to browse the list and select a server to play on, at which point they connect directly to the server.

The Server Browsing SDK manages the entire client-side portion of this process - server list retrieval, server querying, etc. The SDK is a data engine only. You will be responsible for creating all the GUI elements that are required for a server browser in your game. These typically include

buttons, long multi-column lists, scrollbars, and edit controls.

Server Browsing is typically used for matchmaking of "dedicated server" games. Dedicated server games are those in which a stand-alone server is run, and outside clients connect to the server. The server continues running even when no players are connected to it, and typically does not have a local client itself. For games that require a group of people to all join together at once, and that do not have persistent servers running (i.e. they use Peer to Peer networking, or require one of the players to "host" the game) the GameSpy Matchmaking Toolkit - which includes the Peer SDK - may be a better choice.

The Peer SDK extends the features of Server Browsing to include a dynamic list of available games, integrated chat lobbies and staging rooms, and more. The Peer SDK can be used for dedicated-server games as well, but using the Server Browsing SDK is quicker and easier to implement if you are simply looking to provide a list-based matchmaking experience.

Two examples are included for your review.

sbctest is a simple C based server list program that demonstrates how to easily receive and display a server list.

sbmfcsample is a C++ / MFC based server list with a full GUI. Your in-game server browser will most likely look something like this (with the MFC code replaced by your custom GUI code).

This document provides a simple, pseudo-code based set of instructions for implementing an in-game server browser using the Server Browser. Your code will vary based on your specific GUI interfaces.

See the reference documentation for detailed descriptions of each function.

File Manifest

The following files should be included with this package. If any of the files are missing, please [contact devsupport@gamespy.com](mailto:devsupport@gamespy.com).

File	Description
<i>sb_serverbrowsing.h</i>	Main header file for the Server Browsing SDK - includes all public functions
<i>sb_serverbrowsing.c</i>	Code for primary server browsing functionality
<i>sb_serverlist.c</i>	Server list / master server communication code
<i>sb_queryengine.c</i>	Server query engine code
<i>sb_server.c</i>	Functions for manipulating individual server objects
<i>sb_internal.h</i>	Header files for private functions
<i>sb_crypt.c,h</i>	Encryption code used for master server communication
<i>../qr2/qr2regkeys.c,h</i>	Defines for pre-defined key names

In addition, to build the SDK and samples, you will need to separately download the GameSpy "common code" package, which includes the shared SDK code used by this SDK and others.

When extracting this package, make sure you preserve the directory tree in order to assure that the code builds correctly.

Implementation

Step 0: Implement the Query and Reporting 2 SDK

If you haven't already done so, you need to implement the Query and Reporting 2 SDK in your game. The Query and Reporting 2 SDK allows your game server to report to our master server and be listed for clients to query. Although you can test the Server Browsing SDK by querying other games, typically you'll want to do most of your testing with your own game.

Once you have implemented and tested the Query and Reporting 2 SDK, you can continue with the server list implementation.

Step 1: Create A Server Browser

Your first step should be to create a server browser object (not a true C++ object, just in the sense of an abstract type).

You can create the server browser when your game first starts, or right before you update the list. You can update a single server browser as many times as you want, so you probably won't need to create it more than once (although you can).

To create a server browser, call:

```
ServerBrowser ServerBrowserNew(const gsi_char *query
                               const gsi_char *queryFromGamename, const gsi
                               int queryFromVersion, int maxConcUpdates, ir
                               SBBool lanBrowse, ServerBrowserCallback call
```

queryForGamename

The name of the game you want to browse for. [gamenames](#) are issued by our developer relations team. If you do not have one already, contact_devsupport@gamespy.com to request a [gamename](#) for your game. For testing purposes, you may wish to query for a [gamename](#) other than your game - for example, to test how well your list works with several thousand servers listed.

[Contact devsupport@gamespy.com](mailto:devsupport@gamespy.com) for a list of alternative [gamenames](#) you can use for testing.

[queryFromGamename](#)

Your [gamename](#) that you were issued with your secret key. If you don't have a [gamename](#) or secret key, please [contact devsupport@gamespy.com](#). This will typically be the same as [queryForGamename](#), unless you are querying for a different game list during testing.

[queryFromKey](#)

The secret key you were issued that corresponds to your [queryFromGamename](#)

[queryFromVersion](#)

A version identifier for your game - this is optional and should be set to 0 unless you are told otherwise by developer support

[maxConcUpdates](#)

The maximum number of concurrent updates (queries) that will be made. 10 is appropriate for modem users, broadband users can generally accommodate 20-30. You can either present this as an option to your users, or leave it at 10 for everyone. Higher numbers lead to faster refreshes of the server list, but if the refresh speed exceeds the user's capacity, ping times will be measured inaccurately and some servers will time out and not show up on the list.

[queryVersion](#)

Determines the protocol used for server queries. If your game implements the Query and Reporting 2 SDK (as most new games will), simply pass [QVERSION_QR2](#). If you are updating a legacy game to support the Server Browsing SDK, but the game still uses the original Query and Reporting SDK, then you should pass [QVERSION_G0A](#).

[lanBrowse](#)

The switch to turn on only LAN browsing - use [SBTrue](#) if you want only LAN browsing.

[callback](#)

The server browser callback function described in Step 3. For now

you can just create a stub function with a prototype like:

```
void SBCallback(ServerBrowser sb, SBCallbackRea
void *instance)
{ } // todo
```

instance

Any game-specific data you want passed to the callback function. For example, you can pass a structure pointer or object pointer for use within the Callback. If you can access any needed data within the Callback already, then you can just pass `NULL` for instance.

This step should look something like:

```
int CMyGame::OnMultiplayerButtonClicked(...)
{
    m_ServerBrowser = ServerBrowserNew("mygame", "mygame
}
```

Step 2: Update The Server Browser

When you are ready to populate the Server Browser with data, you need to call one of the update functions. There are two update functions available:

ServerBrowserUpdate

Obtains a list of servers from the master server, and then queries the individual servers for information.

ServerListLANUpdate

Scans the local LAN for games and updates the list with them.

Note that if you've already updated the server list once, you need to call `ServerBrowserClear` to clear the list of servers (otherwise you will end up with duplicates). You can call `ServerBrowserClear` even if the list hasn't been updated yet (it won't break anything).

When calling either of the update functions, you need to decide whether you want to do updates Synchronously or Asynchronously.

With a Synchronous update, the Update function does not return until the entire list is finished updating. Your callback function will still be called during the update (so you can add servers to your list or repaint the window) but there is no guarantee as to how often the callback will be called. If you choose to use Synchronous updates, it is important that you display status messages to the user as the list state changes, otherwise they may think the game has locked up.

With Asynchronous updates, the Update function returns immediately, but you are responsible for calling the [ServerBrowserThink](#) function every 10-100 ms while the list is being updated. If you do not call the function, the update will not occur. If you do not call it often enough, the ping times for servers will be inaccurate.

The sbctest sample uses Synchronous updates, while the Sample1 sample uses Asynchronous updates (and a Windows timer to trigger the [ServerBrowserThink](#) function).

You may find it easier to implement the browser with Synchronous updates at first, then switch to Asynchronous updates once it has been fully tested.

The following additional parameters are used with the [ServerBrowserUpdate](#) function:

[disconnectOnComplete](#)

Determines whether the Server Browser disconnects from the master server after obtaining the server list, or stays connected. The only reason to stay connected is if you expect to be querying the master server for full details on individual servers (using [ServerBrowserAuxUpdateServer](#)) in a game that supports NAT Negotiation (and thus may have servers hosted behind firewalls). In most cases you should pass [SBTrue](#) here. Even if you need to contact the master server to obtain information with [ServerBrowserAuxUpdateServer](#), that function will automatically reconnect as-needed.

basicFields

An array of information keys that you want to retrieve from each server in the list.

```
unsigned char basicFields[] = {HOSTNAME_KEY, GA  
int numFields = sizeof(basicFields) / sizeof(ba
```

This means that the hostname, gametype, mapname, numplayers, and maxplayers keys will be retrieved for each server in your list. The key indexes are the same as those used in the QR2 SDK. If you've defined custom key indexes for custom keys in your game, you can include those as well. Note that you **MUST** register all custom keys with `qr2_register_key` before using their indexes in this function. Once you have the basic keys for servers, you can go back and query for the "full" list of keys your game reports - including player and team keys. Details on re-querying servers for additional information is contained in Step 5.

Note that when using the LAN Update function, you do not need to specify a list of keys - LAN updates automatically retrieve all keys and values from the server.

numBasicFields

The number of basic fields passed in your array.

serverFilter

A filter that will be applied on the master server, prior to sending the list of servers to the client. When applied correctly, filtering can dramatically increase the speed of server list updates by reducing the number of servers to query, and can make it easier for players to find the types of game they are looking for.

All of your server keys are available for filtering, as well as two special keys that are added by the master server:

country

the two-letter ISO country code where a server is located - as-determined by the IP address of the server

region

a numeric bitmask that identifies the region a server is

located in (based on the country). See the region list appendix at the end of this document for all available regions.

Note that filtering by "ping" is not possible, since ping is determined by the client - not the master server.

Filter strings are written using SQL-like syntax. Most standard SQL operators are available. Wildcard string comparisons can be made using the "like" operator, with % as the wildcard character.

Here are some examples of useful filter strings:

- *gametype = 'ctf'* Only returns games whose gametype key matches 'ctf'
- *numplayers > 0 and numplayers != maxplayers*
Only returns servers who have players on them, but are not full
- *password = 0*
Only returns servers that are not passworded (assuming you use a "password" key to indicate password protected servers)
- *hostname like '%[gsf]%'*
Only returns servers that have the string '[gsf]' somewhere in their name. Could be used, for example, to allow a player to find just the servers their clan runs.
- *(country = 'DE' or country = 'FR') and maxplayers >= 8*
Only returns servers located in Germany or France that support 8 or more players.

Pass NULL or an empty string if no filtering is required.

For LAN Updates you will also need to specify the ports to check for servers on. The Server Browsing SDK will scan a range of ports by sending a query packet to the broadcast address for each one.

[startSearchport](#) is usually your standard query port (e.g. the UDP port number you pass to [qr2_init](#) in the Query and Reporting SDK).

[endSearchPort](#) is the highest port to scan. When multiple servers are run on the same machine, the QR2 SDK allocates incrementally higher

ports from the starting port for each server. In general, it is not recommended to use an end port more than 100 higher than the start port. A packet must be sent out to all ports between start and end, and higher numbers can lead to broadcast storms.

This step should look like:

```
int CMyGame::OnRefreshInternetButtonClicked(...)
{
    ServerBrowserUpdate(m_ServerBrowser, SBFalse
    ...
}
int CMyGame::OnRefreshLANButtonClicked(...)
{
    ServerBrowserLANUpdate(m_ServerBrowser, SBFal
    ...
}
```

Step 3: Create The List Callback

As the Server Browser updates the server list, it calls back to the function you passed in [ServerBrowserNew](#) to give status and progress updates.

The `sb` parameter is the [ServerBrowser](#) object the callback is referring to.

The `reason` parameter is one of the following values:

[sbc_serveradded](#)

A server was added to the list. Note that you may just have an IP and port at this point - all servers are added before they are queried. You can choose to add the server to your UI at this point, or wait until you get a [serverupdated](#) callback for the server and have more information to display about it.

[sbc_serverupdated](#)

The information for a server has been updated. Either basic or full information is now available about this server.

[sbc_serverupdatefailed](#)

An attempt to retrieve information about this server, either directly or from the master server, failed. The server is down or unreachable.

[sbc_serverdeleted](#)

A server was removed from the list. This only occurs when using push updates, which are not generally used in the Server Browsing SDK (only the Peer SDK).

[sbc_updatecomplete](#)

All queued updates have been completed and the query engine is now idle.

[sbc_queryerror](#)

The master returned an error string for the provided query. Typically due to a filter string syntax error. You can obtain the text of the error message from [ServerBrowserListQueryError](#)

The [server](#) parameter indicates the server that is being referred to, if the message is server-specific.

[instance](#) is the instance pointer you passed in when initializing the Server Browser object.

This step should look something like:

```
void SBCallback(ServerBrowser sb, SBCallbackReason r
{
    CMyGame *g = (CMyGame *)instance;
    switch (reason)
    {
    case sbc_serveradded :
        g->ServerView->AddServerToList(server)
        break;
    case sbc_serverupdated :
        g->ServerView->UpdateServerInList(serv
        break;
    case sbc_updatecomplete:
        g->ServerView->SetStatus("Update Compl
        break;
```

```

        case sbc_queryerror:
            g->ServerView->SetStatus("Query Error Occurred  
ServerBrowserListQueryError(sb));
            break;
        }
    }
}

```

Step 4: Extracting and Displaying Server Information

Somewhere along the line (either in your Callback or a helper function) you will need to actually get the data out of the SBServer object to display it on your list.

The Server Browsing SDK provides 10 functions to help you get the data you need from the [SBServer](#) object:

```

const char *SBServerGetStringValue(SBServer server,
int SBServerGetIntValue(SBServer server, char *key,
double SBServerGetFloatValue(SBServer server, char *k
SBBool SBServerGetBoolValue(SBServer server, char *k

```

The first four functions are used to access the values for server key information. Simply use the key name you registered to access the value for that key on a server. Note that only the basic keys you requested in [ServerBrowserUpdate](#) are available after the initial update. See the next step for information on getting the rest of the keys/values from the server.

```

const char *SBServerGetPlayerStringValue(SBServer se
int SBServerGetPlayerIntValue(SBServer server, int p
double SBServerGetPlayerFloatValue(SBServer server,

```

The second set of functions returns a specific player key. You can get the same result by asking for a server key in the form `keyname_N` where N is the player number you are interested in. The [SBServerGetPlayer](#) functions just provide a shortcut to this. To get the ping for player 0, you would ask for [SBServerGetPlayerIntValue\(server, 0,](#)

the game, and the team information. To obtain this additional information when someone selects a server, you will need to perform what is known as an "Auxiliary" update of the server. To accomplish this, call:

```
SBError ServerBrowserAuxUpdateServer(ServerBrowser s
```

server

The server you want to get updated information for.

async

Determines whether the function returns immediately, or waits for the update to complete. Note that if you perform the [AuxUpdate](#) asynchronously, you must call the [ServerBrowserThink](#) function to perform processing.

fullUpdate

Determines whether basic or full keys are retrieved from the server. Generally you will want to pass [SBTrue](#) to retrieve all the available keys/values from the server.

You should check to see if the server already has full keys available with [SBServerHasFullKeys](#) to avoid updating the server if not needed.

Multiple aux updates can be queued at a time if you want to get full keys for a range of servers.

If the server being updated is behind a NAT, and does not support direct-UDP queries (only for games that use the NAT Negotiation SDK), then the full server information is obtained from the master server, instead of the game server directly. This requires a connection to the master server. If you set the [disconnectOnComplete](#) option when updating, a connection will be re-established to the master to obtain the information.

You can also use the [ServerBrowserAuxUpdateIP](#) function to "add" a server to the list that was not already present. Instead of providing an [SBServer](#) object, you will provide the IP and query port of the server. You can use this to allow players to manually add servers to the list, or to store a list of favorites locally and add them to the list directly.

Step 6: Sorting and Other Features

You will probably want to give players the ability to sort the server list on a specific column (for example, the server name to find a specific server, by ping to find the best server, or by players to find the most crowded). If your list control supports sorting, you can do it that way, or, you can resort the actual `ServerBrowser` object and repopulate your display with the sorted data. Unlike the previous CEngine SDK, list storage is decoupled from updating, so you can use the sorting functions to resort the list while it is being updated (although resorting after every server update arrives may lead to poor performance on large lists).

To resort the internal list, use the `ServerBrowserSort` function:

```
void ServerBrowserSort(ServerBrowser sb, SBBool asce
```

You should pass in whether you want the list to be sorted in ascending or descending order, what key it should be sorted on (e.g. "ping" or "hostname" or "numplayers") and the value type for that key (e.g. `sbcm_int` for integer comparison, `sbcm_stricase` for case-insensitive string comparison).

To sort the list ascending by ping you would call:

```
ServerBrowserSort(sb, SBTrue, "ping", sbcm_int);
```

Once you have resorted the list, you will need to clear your display and repopulate it with the sorted list. This is typically done with a `FOR` loop from 0 to `ServerBrowserCount(...)-1` in which you call `ServerBrowserGetServer` to get each server in the list.

In order to display the progress of the server list update, you can use the `ServerBrowserPendingQueryCount` to determine the number of servers waiting to be queried. By comparing this to the number of servers on the list, you can determine a completion percentage.

The `ServerBrowserHalt` function can be used to stop an update in progress, if the user wants to abort the update.

The `ServerBrowserState` can be used to determine the current state

of the Server Browser. Descriptions of the possible state values can be found in the main header file.

Step 7: Free the Server Browser When Done

When you are completely done with the server browser, call the [ServerBrowserFree](#) function to free the memory allocated by the list. The Server Browser object is invalid after this call, so do not use it again without calling [ServerBrowserNew](#).

UNICODE Support

The GameSpy SDKs support an optional UNICODE interface for widestring applications. To use this interface, first define the symbol [GSI_UNICODE](#). Then, use widestrings wherever ANSI strings were previously called for. When in doubt, please refer to the header files for specific function declarations.

Although the GameSpy SDK interfaces support UNICODE parameters, some items may be stripped of their extra UNICODE information. These items include: nickname, email address, and URL strings. You may pass in widestring values, but they will first be converted to their ANSI counterparts before transmission.

Appendix: Changes From The CEngine SDK

The Server Browsing SDK replaces the CEngine SDK, and provides a number of changes and enhancements. Migration from the CEngine SDK is fairly straight-forward and can provide a number of benefits.

The changes and improvements in the SDK are listed below to help you with migration:

- Defaults to a multi-step update process, where only basic keys are obtained from servers initially, and full keys are obtained on-request. This results in a significant bandwidth reduction for both clients and servers (on the order of 5-10X) and leads to faster refreshes with less overhead.
- Supports the new QR2 querying protocol, which allows for querying of individual keys and optimized encoding of key data (key names are no longer sent if not required).
- Allows server information to be obtained from the master server for games hosted behind a NAT/Firewall.
- Supports the new NAT Negotiation SDK for hosting and connecting to games behind a NAT or Firewall
- Allows for filtering by any server key, instead of the fixed list of keys available for filtering in the CEngine SDK
- Allows for filtering of servers by country and region, without requiring server administrators to report that information.
- Allows for sorting of the server list while updates are still in progress.
- Faster socket code that uses a single UDP socket for all queries, instead of requiring 1 socket for each simultaneous query.
- Server lists from the master server use a new format that is compressed even more than before.

Appendix: Region Codes and Usage

The updated Master Server backend that supports the Server Browsing SDK has the ability to identify the country a server is located in based on its IP address, and based on that country, sort it into a specific region. Master regions are made up of smaller regions. You can use filtering to restrict the list of servers returned to a specific country or region, thus giving players a list that better represents their play-able servers.

Regions are identified by a region ID number, however a particular server can be listed in multiple regions since regions are "nested". Because of this, the region number for a server is actually the sum of all the regions the server is in. To filter on a specific region, you should use the bitwise-AND operator to identify servers that are listed in that region.

For example, to identify servers in North America, you would use the filter: $(region \& 2) = 2$, since 2 is the region code for North America. Normal bitwise math can be used to check for multiple regions. For example, to check for North America or Caribbean, you can add them together: $(region \& 6) \neq 0$

regionid	regionname
1	Americas
2	North America
4	Caribbean
8	Central America
16	South America
32	Africa
64	Central Africa
128	East Africa
256	Northern Africa
512	Southern Africa
1024	West Africa
2048	Asia
4096	East Asia
8192	Pacific
16384	South Asia
32768	South-East Asia
65536	Europe

131072	Baltic States
262144	Commonwealth of Independent States
524288	Eastern Europe
1048576	Middle East
2097152	South-East Europe
4194304	Western Europe

ccode	country	regionname
-----	-----	-----
BI	Burundi	Central Africa
CM	Cameroon	Central Africa
CF	Central African Republic	Central Africa
TD	Chad	Central Africa
CG	Congo	Central Africa
GQ	Equatorial Guinea	Central Africa
RW	Rwanda	Central Africa
DJ	Djibouti	East Africa
ER	Eritrea	East Africa
ET	Ethiopia	East Africa
KE	Kenya	East Africa
SC	Seychelles	East Africa
SO	Somalia	East Africa
SH	St. Helena	East Africa
SD	Sudan	East Africa
TZ	Tanzania	East Africa
UG	Uganda	East Africa
DZ	Algeria	Northern Africa
EG	Egypt	Northern Africa
LY	Libya	Northern Africa
MA	Morocco	Northern Africa
TN	Tunisia	Northern Africa
AO	Angola	Southern Africa
BW	Botswana	Southern Africa
BV	Bouvet Island	Southern Africa
KM	Comoros	Southern Africa
HM	Heard and McDonald Islands	Southern Africa
LS	Lesotho	Southern Africa
MG	Madagascar	Southern Africa
MW	Malawi	Southern Africa
MU	Mauritius	Southern Africa
YT	Mayotte	Southern Africa
MZ	Mozambique	Southern Africa
NA	Namibia	Southern Africa
RE	Reunion	Southern Africa
ZA	South Africa	Southern Africa
SZ	Swaziland	Southern Africa
ZM	Zambia	Southern Africa
ZW	Zimbabwe	Southern Africa
BJ	Benin	West Africa
BF	Burkina Faso	West Africa

CV	Cape Verde	West Africa
CI	Cote D`ivoire	West Africa
GA	Gabon	West Africa
GM	Gambia	West Africa
GH	Ghana	West Africa
GN	Guinea	West Africa
GW	Guinea-Bissau	West Africa
LR	Liberia	West Africa
ML	Mali	West Africa
MR	Mauritania	West Africa
NE	Niger	West Africa
NG	Nigeria	West Africa
ST	Sao Tome and Principe	West Africa
SN	Senegal	West Africa
SL	Sierra Leone	West Africa
TG	Togo	West Africa
AI	Anguilla	Caribbean
AG	Antigua and Barbuda	Caribbean
AW	Aruba	Caribbean
BS	Bahamas	Caribbean
BB	Barbados	Caribbean
BM	Bermuda	Caribbean
KY	Cayman Islands	Caribbean
CU	Cuba	Caribbean
DM	Dominica	Caribbean
DO	Dominican Republic	Caribbean
GD	Grenada	Caribbean
GP	Guadeloupe	Caribbean
HT	Haiti	Caribbean
JM	Jamaica	Caribbean
MQ	Martinique	Caribbean
MS	Montserrat	Caribbean
AN	Netherlands Antilles	Caribbean
PR	Puerto Rico	Caribbean
VC	Saint Vincent and The Grenadin	Caribbean
KN	St Kitts-Nevis	Caribbean
LC	St Lucia	Caribbean
TT	Trinidad & Tobago	Caribbean
TC	Turks & Caicos Islands	Caribbean
VG	Virgin Islands (British)	Caribbean
VI	Virgin Islands (US)	Caribbean
BZ	Belize	Central America
CR	Costa Rica	Central America
SV	El Salvador	Central America
GT	Guatemala	Central America
HN	Honduras	Central America
MX	Mexico	Central America
NI	Nicaragua	Central America
PA	Panama	Central America
CA	Canada	North America

GL	Greenland	North America
PM	St. Pierre and Miquelon	North America
US	United States	North America
UM	US Minor Outlying Islands	North America
AR	Argentina	South America
BO	Bolivia	South America
BR	Brazil	South America
CL	Chile	South America
CO	Colombia	South America
EC	Ecuador	South America
FK	Falkland Islands (Malvinas)	South America
GF	French Guiana	South America
GY	Guyana	South America
PY	Paraguay	South America
PE	Peru	South America
GS	S. Georgia and S. Sandwich Isl	South America
SR	Suriname	South America
UY	Uruguay	South America
VE	Venezuela	South America
CN	China	East Asia
HK	Hong Kong	East Asia
JP	Japan	East Asia
MO	Macao	East Asia
MN	Mongolia	East Asia
KP	North Korea	East Asia
KR	South Korea	East Asia
TW	Taiwan	East Asia
AS	American Samoa	Pacific
AU	Australia	Pacific
CK	Cook Islands	Pacific
FJ	Fiji	Pacific
PF	French Polynesia	Pacific
GU	Guam	Pacific
KI	Kiribati	Pacific
MH	Marshall Islands	Pacific
FM	Micronesia	Pacific
NR	Nauru	Pacific
NC	New Caledonia	Pacific
NZ	New Zealand	Pacific
NU	Niue	Pacific
NF	Norfolk Island	Pacific
MP	Northern Mariana Islands	Pacific
PG	Papua New Guinea	Pacific
PN	Pitcairn Islands	Pacific
EH	Samoa	Pacific
SB	Solomon Islands	Pacific
TO	Tonga	Pacific
TK	Tonga	Pacific
TV	Tuvalu	Pacific
VU	Vanuatu	Pacific

WF	Wallis and Futuna Islands	Pacific
AF	Afghanistan	South Asia
BD	Bangladesh	South Asia
BT	Bhutan	South Asia
IO	British Indian Ocean Territory	South Asia
IN	India	South Asia
MV	Maldives	South Asia
NP	Nepal	South Asia
PK	Pakistan	South Asia
LK	Sri Lanka	South Asia
BN	Brunei Darussalam	South-East Asia
KH	Cambodia	South-East Asia
CX	Christmas Island	South-East Asia
CC	Cocos (Keeling Islands)	South-East Asia
TP	East Timor	South-East Asia
ID	Indonesia	South-East Asia
LA	Laos	South-East Asia
MY	Malaysia	South-East Asia
MM	Myanmar	South-East Asia
PW	Palau	South-East Asia
PH	Philippines	South-East Asia
SG	Singapore	South-East Asia
TH	Thailand	South-East Asia
VN	Vietnam	South-East Asia
EE	Estonia	Baltic States
LV	Latvia	Baltic States
LT	Lithuania	Baltic States
AM	Armenia	CIS
AZ	Azerbaijan	CIS
BY	Belarus	CIS
GE	Georgia	CIS
KZ	Kazakhstan	CIS
KG	Kyrgyzstan	CIS
MD	Moldova	CIS
RU	Russian Federation	CIS
TJ	Tajikistan	CIS
TM	Turkmenistan	CIS
UA	Ukraine	CIS
UZ	Uzbekistan	CIS
CZ	Czech Republic	Eastern Europe
HU	Hungary	Eastern Europe
PL	Poland	Eastern Europe
RO	Romania	Eastern Europe
SK	Slovak Republic	Eastern Europe
BH	Bahrain	Middle East
IR	Iran	Middle East
IQ	Iraq	Middle East
IL	Israel/Occupied Territories	Middle East
JO	Jordan	Middle East
KW	Kuwait	Middle East

LB	Lebanon	Middle East
OM	Oman	Middle East
QA	Qatar	Middle East
SA	Saudi Arabia	Middle East
SY	Syria	Middle East
AE	United Arab Emirates	Middle East
YE	Yemen	Middle East
AL	Albania	South-East Europe
BA	Bosnia-Herzegovina	South-East Europe
BG	Bulgaria	South-East Europe
HR	Croatia	South-East Europe
CY	Cyprus	South-East Europe
GR	Greece	South-East Europe
MK	Macedonia	South-East Europe
MT	Malta	South-East Europe
SI	Slovenia	South-East Europe
TR	Turkey	South-East Europe
YU	Yugoslavia	South-East Europe
AD	Andorra	Western Europe
AT	Austria	Western Europe
BE	Belgium	Western Europe
DK	Denmark	Western Europe
FO	Faroe Islands	Western Europe
FI	Finland	Western Europe
FR	France	Western Europe
DE	Germany	Western Europe
GI	Gibraltar	Western Europe
IS	Iceland	Western Europe
IE	Ireland	Western Europe
IT	Italy	Western Europe
LI	Liechtenstein	Western Europe
LU	Luxembourg	Western Europe
MC	Monaco	Western Europe
NL	Netherlands	Western Europe
NO	Norway	Western Europe
PT	Portugal	Western Europe
SM	San Marino	Western Europe
ES	Spain	Western Europe
SJ	Svalbard and Jan Mayen Islands	Western Europe
SE	Sweden	Western Europe
CH	Switzerland	Western Europe
UK	United Kingdom	Western Europe
VA	Vatican	Western Europe

Server Browsing SDK Functions

[SBServerDirectConnect](#)

Indicates whether the server supports direct UDP connections.

[SBServerEnumKeys](#)

Enumerates the keys/values for a given server by calling KeyEnumFn with each key/value. The user-defined instance data will be passed to the KeyFn callback.

[SBServerGetBoolValue](#)

Returns the value associated with the specified key. This value is returned as the appropriate type: SBBool, float, int or string.

[SBServerGetConnectionInfo](#)

Checks if Nat Negotiation is required, based off whether it is a LAN game, a public IP is present and several other factors. Fills a supplied pointer with an IP string to use for Nat Negotiation, or a direct connection if possible.

[SBServerGetFloatValue](#)

Returns the value associated with the specified key. This value is returned as the appropriate type: SBBool, float, int or string.

[SBServerGetIntValue](#)

Returns the value associated with the specified key. This value is returned as the appropriate type: SBBool, float, int or string.

[SBServerGetPing](#)

Returns the stored ping time for the specified server.

[SBServerGetPlayerFloatValue](#)

Returns the value associated with the specified player's key. This value is returned as the appropriate type. Float, int or string.

[SBServerGetPlayerIntValue](#)

Returns the value associated with the specified player's key. This value is returned as the appropriate

type. Float, int or string.

[SBServerGetPlayerStringValue](#)

Returns the value associated with the specified player's key. This value is returned as the appropriate type. Float, int or string.

[SBServerGetPrivateAddress](#)

Returns the internal address of the SBServer, if any. For users behind a NAT or firewall, this is the local DHCP or assigned IP address of the machine.

[SBServerGetPrivateInetAddress](#)

Returns the internal address of the SBServer, if any. For users behind a NAT or firewall, this is the local DHCP or assigned IP address of the machine.

[SBServerGetPrivateQueryPort](#)

Returns the private query port of the specified server. This is the internal port on which the server communicates to the

GameSpy backend.

[SBServerGetPublicAddress](#)

Returns the external address of the SBServer, if any. For users behind a NAT or firewall, this is the address of the outermost NAT or firewall layer.

[SBServerGetPublicInetAddress](#)

Returns the external address of the SBServer, if any. For users behind a NAT or firewall, this is the address of the outermost NAT or firewall layer.

[SBServerGetPublicQueryPort](#)

Returns the public query port of the specified server. This is the external port on which the GameSpy backend communicates with the server.

[SBServerGetStringValue](#)

Returns the value associated with the specified key. This value is returned as the appropriate type. SBBool, float, int or

	string.
SBServerGetTeamFloatValue	Returns the value associated with the specified teams' key. This value is returned as the appropriate type; Float, int or string.
SBServerGetTeamIntValue	Returns the value associated with the specified teams' key. This value is returned as the appropriate type; Float, int or string.
SBServerGetTeamStringValue	Returns the value associated with the specified teams' key. This value is returned as the appropriate type; Float, int or string.
SBServerHasBasicKeys	Determine if basic information is available for the specified server.
SBServerHasFullKeys	Determine if full information is available for the specified

	server.
SBServerHasPrivateAddress	Tests to see if a private address is available for the server.
SBServerHasValidPing	Determines if a server has a valid ping value (otherwise the ping will be 0).
ServerBrowserAuxUpdateIP	Queries key/values from a single server.
ServerBrowserAuxUpdateServer	Query key/values from a single server that has already been added to the internal list.
ServerBrowserClear	Clear the current server list.
ServerBrowserConnectToServer	Connects to a game server.
ServerBrowserCount	Retrieves the current list of games from the GameSpy master server.
ServerBrowserDisconnect	Disconnect from the

	GameSpy master server.
ServerBrowserErrorDesc	Returns a human readable string for the specified SBError.
ServerBrowserFree	Frees memory allocated by the ServerBrowser SDK. Terminates any pending queries.
ServerBrowserGetMyPublicIP	Returns the local client's external (firewall) address.
ServerBrowserGetMyPublicIPAddr	Returns the local client's external (firewall) address.
ServerBrowserGetServer	Returns the SBServer object at the specified index.
ServerBrowserGetServerByIP	Returns the SBServer with the specified IP
ServerBrowserHalt	Stop an update in progress.
ServerBrowserLANSetLocalAddr	

	Sets the network adapter to use for LAN broadcasts (optional)
ServerBrowserLANUpdate	Retrieves the current list of games broadcasting on the local network.
ServerBrowserLimitUpdate	Retrieves the current limited list of games from the GameSpy master server. Useful for low-memory systems.
ServerBrowserListQueryError	Returns the ServerList error string, if any.
ServerBrowserNew	Initialize the ServerBrowser SDK.
ServerBrowserPendingQueryCount	Retrieves the number of servers with outstanding queries. Use this to check progress while asynchronously updating the server list.
ServerBrowserRemoveIP	Removes a server from the local list.

ServerBrowserRemoveServer	Removes a server from the local list.
ServerBrowserSendMessageToServer	Sends a game specific message to the specified IP/port. This message is routed through the master server.
ServerBrowserSendNatNegotiateCookieToServer	Sends a nat negotiation cookie to the server. The cookie is sent via the master server.
ServerBrowserSort	Sort the current list of servers.
ServerBrowserState	Gets current state of the Server Browser object.
ServerBrowserThink	Allows ServerBrowsingSDK to continue internal processing including processing query replies.
ServerBrowserUpdate	Retrieves the current list of games from the

GameSpy master
server.

SBServerDirectConnect

Indicates whether the server supports direct UDP connections.

```
SBBool SBServerDirectConnect(  
    SBServer server );
```

Routine	Required Header	Distribution
SBServerDirectConnect	<sb_serverbrowsing.h>	SDKZIP

Return Value

Returns SBTrue if a direct connection is possible, otherwise SBFalse.

Parameters

server

[in] A valid SBServer object.

Remarks

A return of SBFalse usually means that NatNegotiation is required.

Section Reference: [Gamespy Server Browsing SDK](#)

See Also: [ServerBrowserNew](#), [ServerBrowserUpdate](#),
[ServerBrowserGetServer](#)

SBServerEnumKeys

Enumerates the keys/values for a given server by calling KeyEnumFn with each key/value. The user-defined instance data will be passed to the KeyFn callback.

```
void SBServerEnumKeys(  
    SBServer server,  
    SBServerKeyEnumFn KeyFn,  
    void * instance );
```

Routine	Required Header	Distribution
SBServerEnumKeys	<sb_serverbrowsing.h>	SDKZIP

Parameters

server

[in] A valid SBServer object.

KeyFn

[in] A callback that is called once for each key.

instance

[in] A user-defined data value that will be passed into each call to KeyFn.

Remarks

The **SBServerEnumKeys** function is used to list the available keys for a particular SBServer object. This is often useful when the number of keys or custom keys is unknown or variable. Most often, the number of keys is predefined and constant, making this function call unnecessary. No query is sent when enumerating keys, instead the keys are stored from the previous server update.

Section Reference: [Gamespy Server Browsing SDK](#)

See Also: [ServerBrowserNew](#), [ServerBrowserUpdate](#), [ServerBrowserGetServer](#)

SBServerGetBoolValue

Returns the value associated with the specified key. This value is returned as the appropriate type: SBBool, float, int or string.

```
SBBool SBServerGetBoolValue(  
    SBServer server,  
    const gsi_char * key,  
    SBBool bdefault );
```

Routine	Required Header	Distribution
SBServerGetBoolValue	<sb_serverbrowsing.h>	SDKZIP

Return Value

If the key is invalid or missing, the specified default is returned. For an existing key, the value is converted from string form to the appropriate data type. These functions do not perform any type checking.

Parameters

server

[in] A valid SBServer object.

key

[in] The value associated with this key will be returned.

bdefault

[in] The value to return if the key is not found.

Remarks

These functions are usefull for converting custom keys to a native data type. No type checking is performed, the string value is simply cast to the appropriate data type. If a key is not found,the supplied default is returned.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
SBServerGetBoolValue	SBServerGetBoolValueA	SBServerGetBoolValueW

SBServerGetBoolValueW and **SBServerGetBoolValueA** are UNICODE and ANSI mapped versions of **SBServerGetBoolValue**. The arguments of **SBServerGetBoolValueA** are ANSI strings; those of **SBServerGetBoolValueW** are wide-character strings.

Section Reference: [Gamespy Server Browsing SDK](#)

See Also: [ServerBrowserNew](#), [ServerBrowserUpdate](#), [ServerBrowserGetServer](#)

SBServerGetConnectionInfo

Checks if Nat Negotiation is required, based off whether it is a LAN game, a public IP is present and several other factors. Fills a supplied pointer with an IP string to use for Nat Negotiation, or a direct connection if possible.

```
SBBool SBServerGetConnectionInfo(  
    ServerBrowser gSB,  
    SBServer server,  
    gsi_u16 portToConnectTo,  
    char * ipstring_out );
```

Routine	Required Header	Distribution
SBServerGetConnectionInfo	<sb_serverbrowsing.h>	SDKZIP

Return Value

Returns SBTrue if Nat Negotiation is required, SBFalse if not.

Parameters

gSB

[in] ServerBrowser object returned from ServerBrowserNew.

server

[in] A valid SBServer object.

portToConnectTo

[in] The game port to connect to.

ipstring_out

[out] An IP String you can use for a direct connection, or to attempt Nat Negotiation with.

Remarks

The connection test will result in one of three scenarios, based upon the return value of the function.

Returns SBFalse:

- 1) LAN game - connect using the IP string.
- 2) Internet game with a direct connection available - connect using the IP string.

Returns SBTrue:

- 3) Nat traversal required, perform Nat Negotiation with the IP string before connecting.

Section Reference: [Gamespy Server Browsing SDK](#)

SBServerGetFloatValue

Returns the value associated with the specified key. This value is returned as the appropriate type: SBBool, float, int or string.

```
double SBServerGetFloatValue(  
    SBServer server,  
    const gsi_char * key,  
    double fdefault );
```

Routine	Required Header	Distribution
SBServerGetFloatValue	<sb_serverbrowsing.h>	SDKZIP

Return Value

If the key is invalid or missing, the specified default is returned. For an existing key, the value is converted from string form to the appropriate data type. These functions do not perform any type checking.

Parameters

server

[in] A valid SBServer object.

key

[in] The value associated with this key will be returned.

fdefault

[in] The value to return if the key is not found.

Remarks

These functions are usefull for converting custom keys to a native data type. No type checking is performed, the string value is simply cast to the appropriate data type. If a key is not found,the supplied default is returned.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
SBServerGetFloatValue	SBServerGetFloatValueA	SBServerGetFloatValueW

SBServerGetFloatValueW and **SBServerGetFloatValueA** are UNICODE and ANSI mapped versions of **SBServerGetFloatValue**. The arguments of **SBServerGetFloatValueA** are ANSI strings; those of **SBServerGetFloatValueW** are wide-character strings.

Section Reference: [Gamespy Server Browsing SDK](#)

See Also: [ServerBrowserNew](#), [ServerBrowserUpdate](#), [ServerBrowserGetServer](#)

SBServerGetIntValue

Returns the value associated with the specified key. This value is returned as the appropriate type: SBBool, float, int or string.

```
int SBServerGetIntValue(  
    SBServer server,  
    const gsi_char * key,  
    int idefault );
```

Routine	Required Header	Distribution
SBServerGetIntValue	<sb_serverbrowsing.h>	SDKZIP

Return Value

If the key is invalid or missing, the specified default is returned. For an existing key, the value is converted from string form to the appropriate data type. These functions do not perform any type checking.

Parameters

server

[in] A valid SBServer object.

key

[in] The value associated with this key will be returned.

idefault

[in] The value to return if the key is not found.

Remarks

These functions are usefull for converting custom keys to a native data type. No type checking is performed, the string value is simply cast to the appropriate data type. If a key is not found,the supplied default is returned.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
SBServerGetIntValue	SBServerGetIntValueA	SBServerGetIntValueW

SBServerGetIntValueW and **SBServerGetIntValueA** are UNICODE and ANSI mapped versions of **SBServerGetIntValue**. The arguments of **SBServerGetIntValueA** are ANSI strings; those of **SBServerGetIntValueW** are wide-character strings.

Section Reference: [Gamespy Server Browsing SDK](#)

See Also: [ServerBrowserNew](#), [ServerBrowserUpdate](#), [ServerBrowserGetServer](#)

SBServerGetPing

Returns the stored ping time for the specified server.

```
int SBServerGetPing(  
    SBServer server );
```

Routine	Required Header	Distribution
SBServerGetPing	<sb_serverbrowsing.h>	SDKZIP

Return Value

The stored server response time.

Parameters

server

[in] A valid SBServer object.

Remarks

The **SBServerGetPing** function will return the stored response time of the server. This response time is calculated from the last server update. Servers behind a firewall will return a ping time of 0. This is because no actual query was sent.

Section Reference: [Gamespy Server Browsing SDK](#)

See Also: [ServerBrowserNew](#), [ServerBrowserUpdate](#), [ServerBrowserGetServer](#)

SBServerGetPlayerFloatValue

Returns the value associated with the specified player's key. This value is returned as the appropriate type. Float, int or string.

```
double SBServerGetPlayerFloatValue(  
    SBServer server,  
    int playernum,  
    const gsi_char * key,  
    double fdefault );
```

Routine	Required Header	Distribution
SBServerGetPlayerFloatValue	<sb_serverbrowsing.h>	SDKZIP

Return Value

If the player or key is invalid or missing, the specified default is returned. For an existing key, the value is converted from string form to the appropriate data type. These functions do not perform any type checking.

Parameters

server

[in] A valid SBServer object.

playernum

[in] The zero based index for the desired player.

key

[in] The value associated with this key will be returned.

fdefault

[in] The value to return if the key is not found.

Remarks

These functions are usefull for converting custom player keys to a native data type. No type checking is performed, the string value is simply cast to the appropriate data type. If a key is not found,the supplied default is returned.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE
SBServerGetPlayerFloatValue	SBServerGetPlayerFloatValueA	SBServer

SBServerGetPlayerFloatValueW and **SBServerGetPlayerFloatValueA** are UNICODE and ANSI mapped versions of **SBServerGetPlayerFloatValue**. The arguments of **SBServerGetPlayerFloatValueA** are ANSI strings; those of **SBServerGetPlayerFloatValueW** are wide-character strings.

Section Reference: [Gamespy Server Browsing SDK](#)

See Also: [ServerBrowserNew](#), [ServerBrowserUpdate](#), [ServerBrowserGetServer](#)

SBServerGetPlayerIntValue

Returns the value associated with the specified player's key. This value is returned as the appropriate type. Float, int or string.

```
int SBServerGetPlayerIntValue(  
    SBServer server,  
    int playernum,  
    const gsi_char * key,  
    int idefault );
```

Routine	Required Header	Distribution
SBServerGetPlayerIntValue	<sb_serverbrowsing.h>	SDKZIP

Return Value

If the player or key is invalid or missing, the specified default is returned. For an existing key, the value is converted from string form to the appropriate data type. These functions do not perform any type checking.

Parameters

server

[in] A valid SBServer object.

playernum

[in] The zero based index for the desired player.

key

[in] The value associated with this key will be returned.

idefault

[in] The value to return if the key is not found.

Remarks

These functions are usefull for converting custom player keys to a native data type. No type checking is performed, the string value is simply cast to the appropriate data type. If a key is not found,the supplied default is returned.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE
SBServerGetPlayerIntValue	SBServerGetPlayerIntValueA	SBServerGetP

SBServerGetPlayerIntValueW and **SBServerGetPlayerIntValueA** are UNICODE and ANSI mapped versions of **SBServerGetPlayerIntValue**. The arguments of **SBServerGetPlayerIntValueA** are ANSI strings; those of **SBServerGetPlayerIntValueW** are wide-character strings.

Section Reference: [Gamespy Server Browsing SDK](#)

See Also: [ServerBrowserNew](#), [ServerBrowserUpdate](#), [ServerBrowserGetServer](#)

SBServerGetPlayerStringValue

Returns the value associated with the specified player's key. This value is returned as the appropriate type. Float, int or string.

```
const gsi_char * SBServerGetPlayerStringValue(  
    SBServer server,  
    int playernum,  
    const gsi_char * key,  
    const gsi_char * sdefault );
```

Routine	Required Header	Distribution
SBServerGetPlayerStringValue	<sb_serverbrowsing.h>	SDKZIP

Return Value

If the player or key is invalid or missing, the specified default is returned. For an existing key, the value is converted from string form to the appropriate data type. These functions do not perform any type checking.

Parameters

server

[in] A valid SBServer object.

playernum

[in] The zero based index for the desired player.

key

[in] The value associated with this key will be returned.

sdefault

[in] The value to return if the key is not found.

Remarks

These functions are usefull for converting custom player keys to a native data type. No type checking is performed, the string value is simply cast to the appropriate data type. If a key is not found,the supplied default is returned.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UN
SBServerGetPlayerStringValue	SBServerGetPlayerStringValueA	SBServe

SBServerGetPlayerStringValueW and **SBServerGetPlayerStringValueA** are UNICODE and ANSI mapped versions of **SBServerGetPlayerStringValue**. The arguments of **SBServerGetPlayerStringValueA** are ANSI strings; those of **SBServerGetPlayerStringValueW** are wide-character strings.

Section Reference: [Gamespy Server Browsing SDK](#)

See Also: [ServerBrowserNew](#), [ServerBrowserUpdate](#), [ServerBrowserGetServer](#)

SBServerGetPrivateAddress

Returns the internal address of the SBServer, if any. For users behind a NAT or firewall, this is the local DHCP or assigned IP address of the machine.

```
char * SBServerGetPrivateAddress(  
    SBServer server );
```

Routine	Required Header	Distribution
SBServerGetPrivateAddress	<sb_serverbrowsing.h>	SDKZIP

Return Value

The private address of the SBServer, in string or integer form.

Parameters

server

[in] A valid SBServer object.

Remarks

When a client machine is behind a NAT or Firewall device, communication must go through the public address. The private address may be used by clients behind the same NAT or Firewall, and may be used to specifically identify two clients with the same public address. Often the private address is of the form “192.168.##.###” and is not usable for communication outside the local network.

The SBServer object may be obtained during the SBCallback from ServerBrowserUpdate, or by calling ServerBrowserGetServer. An SBServer object will only exist for servers in the list. IP addresses removed from the server list will not have an SBServer object associated.

Section Reference: [Gamespy Server Browsing SDK](#)

See Also: [ServerBrowserNew](#), [ServerBrowserUpdate](#), [ServerBrowserGetServer](#)

SBServerGetPrivateInetAddress

Returns the internal address of the SBServer, if any. For users behind a NAT or firewall, this is the local DHCP or assigned IP address of the machine.

```
unsigned int SBServerGetPrivateInetAddress(  
    SBServer server );
```

Routine	Required Header	Distribution
SBServerGetPrivateInetAddress	<sb_serverbrowsing.h>	SDKZIP

Return Value

The private address of the SBServer, in string or integer form.

Parameters

server

[in] A valid SBServer object.

Remarks

When a client machine is behind a NAT or Firewall device, communication must go through the public address. The private address may be used by clients behind the same NAT or Firewall, and may be used to specifically identify two clients with the same public address. Often the private address is of the form "192.168.##.###" and is not usable for communication outside the local network.

The SBServer object may be obtained during the SBCallback from ServerBrowserUpdate, or by calling ServerBrowserGetServer. An SBServer object will only exist for servers in the list. IP addresses removed from the server list will not have an SBServer object associated.

Section Reference: [Gamespy Server Browsing SDK](#)

See Also: [ServerBrowserNew](#), [ServerBrowserUpdate](#), [ServerBrowserGetServer](#)

SBServerGetPrivateQueryPort

Returns the private query port of the specified server. This is the internal port on which the server communicates to the GameSpy backend.

```
unsigned short SBServerGetPrivateQueryPort(  
SBServer server );
```

Routine	Required Header	Distribution
SBServerGetPrivateQueryPort	<sb_serverbrowsing.h>	SDKZIP

Return Value

The private query port.

Parameters

server

[in] A valid SBServer object.

Remarks

The **SBServerGetPrivateQueryPort** function will return the private query port of the server.

Section Reference: [Gamespy Server Browsing SDK](#)

See Also: [ServerBrowserNew](#), [ServerBrowserUpdate](#), [ServerBrowserGetServer](#)

SBServerGetPublicAddress

Returns the external address of the SBServer, if any. For users behind a NAT or firewall, this is the address of the outermost NAT or firewall layer.

```
char * SBServerGetPublicAddress(  
    SBServer server );
```

Routine	Required Header	Distribution
SBServerGetPublicAddress	<sb_serverbrowsing.h>	SDKZIP

Return Value

The public address of the SBServer, in string or integer form.

Parameters

server

[in] A valid SBServer object.

Remarks

When a client machine is behind a NAT or Firewall device, communication must go through the public address. The public address of the SBServer is the address of the outermost Firewall or NAT device.

The SBServer object may be obtained during the SBCallback from ServerBrowserUpdate, or by calling ServerBrowserGetServer. An SBServer object will only exist for servers in the list. IP addresses removed from the server list will not have an SBServer object associated.

Section Reference: [Gamespy Server Browsing SDK](#)

See Also: [ServerBrowserNew](#), [ServerBrowserUpdate](#), [ServerBrowserGetServer](#)

SBServerGetPublicInetAddress

Returns the external address of the SBServer, if any. For users behind a NAT or firewall, this is the address of the outermost NAT or firewall layer.

```
unsigned int SBServerGetPublicInetAddress(
    SBServer server );
```

Routine	Required Header	Distribution
SBServerGetPublicInetAddress	<sb_serverbrowsing.h>	SDKZIP

Return Value

The public address of the SBServer, in string or integer form.

Parameters

server

[in] A valid SBServer object.

Remarks

When a client machine is behind a NAT or Firewall device, communication must go through the public address. The public address of the SBServer is the address of the outermost Firewall or NAT device.

The SBServer object may be obtained during the SBCallback from ServerBrowserUpdate, or by calling ServerBrowserGetServer. An SBServer object will only exist for servers in the list. IP addresses removed from the server list will not have an SBServer object associated.

Section Reference: [Gamespy Server Browsing SDK](#)

See Also: [ServerBrowserNew](#), [ServerBrowserUpdate](#), [ServerBrowserGetServer](#)

SBServerGetPublicQueryPort

Returns the public query port of the specified server. This is the external port on which the GameSpy backend communicates with the server.

```
unsigned short SBServerGetPublicQueryPort(
    SBServer server );
```

Routine	Required Header	Distribution
SBServerGetPublicQueryPort	<sb_serverbrowsing.h>	SDKZIP

Return Value

The public query port.

Parameters

server

[in] A valid SBServer object.

Remarks

The **SBServerGetPublicQueryPort** function will return the public query port of the server.

Section Reference: [Gamespy Server Browsing SDK](#)

See Also: [ServerBrowserNew](#), [ServerBrowserUpdate](#), [ServerBrowserGetServer](#)

SBServerGetStringValue

Returns the value associated with the specified key. This value is returned as the appropriate type. SBBool, float, int or string.

```
const gsi_char * SBServerGetStringValue(  
    SBServer server,  
    const gsi_char * keyname,  
    const gsi_char * def );
```

Routine	Required Header	Distribution
SBServerGetStringValue	<sb_serverbrowsing.h>	SDKZIP

Return Value

If the key is invalid or missing, the specified default is returned. For an existing key, the value is converted from string form to the appropriate data type. These functions do not perform any type checking.

Parameters

server

[in] A valid SBServer object.

keyname

[in] The value associated with this key will be returned.

def

[in] The value to return if the key is not found.

Remarks

These functions are usefull for converting custom keys to a native data type. No type checking is performed, the string value is simply cast to the appropriate data type. If a key is not found,the supplied default is returned.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defini
SBServerGetStringValue	SBServerGetStringValueA	SBServerGetStringV

SBServerGetStringValueW and **SBServerGetStringValueA** are UNICODE and ANSI mapped versions of **SBServerGetStringValue**. The arguments of **SBServerGetStringValueA** are ANSI strings; those of **SBServerGetStringValueW** are wide-character strings.

Section Reference: [Gamespy Server Browsing SDK](#)

See Also: [ServerBrowserNew](#), [ServerBrowserUpdate](#), [ServerBrowserGetServer](#)

SBServerGetTeamFloatValue

Returns the value associated with the specified teams' key. This value is returned as the appropriate type; Float, int or string.

```
double SBServerGetTeamFloatValue(  
    SBServer server,  
    int teamnum,  
    const gsi_char * key,  
    double fdefault );
```

Routine	Required Header	Distribution
SBServerGetTeamFloatValue	<sb_serverbrowsing.h>	SDKZIP

Return Value

If the key is invalid or missing, the specified default is returned. For an existing key, the value is converted from string form to the appropriate data type. These functions do not perform any type checking.

Parameters

server

[in] A valid SBServer object.

teamnum

[in] The integer index of the team.

key

[in] The value associated with this key will be returned.

fdefault

[in] The value to return if the key is not found.

Remarks

These functions are useful for converting custom keys to a native data type. No type checking is performed, the string value is simply cast to the appropriate data type. If a key is not found, the supplied default is returned.

The SBServer object may be obtained during the SBCallback from ServerBrowserUpdate, or by calling ServerBrowserGetServer. An SBServer object will only exist for servers in the list. IP addresses removed from the server list will not have an SBServer object associated.

Team indexes are determined on a per-game basis. The only requirement is that they match the server's reporting indexes.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICC
SBServerGetTeamFloatValue	SBServerGetTeamFloatValueA	SBServerG

SBServerGetTeamFloatValueW and **SBServerGetTeamFloatValueA** are UNICODE and ANSI mapped versions of **SBServerGetTeamFloatValue**. The arguments of **SBServerGetTeamFloatValueA** are ANSI strings; those of **SBServerGetTeamFloatValueW** are wide-character strings.

Section Reference: [Gamespy Server Browsing SDK](#)

See Also: [ServerBrowserNew](#), [ServerBrowserUpdate](#), [ServerBrowserGetServer](#)

SBServerGetTeamIntValue

Returns the value associated with the specified teams' key. This value is returned as the appropriate type; Float, int or string.

```
int SBServerGetTeamIntValue(  
    SBServer server,  
    int teamnum,  
    const gsi_char * key,  
    int idefault );
```

Routine	Required Header	Distribution
SBServerGetTeamIntValue	<sb_serverbrowsing.h>	SDKZIP

Return Value

If the key is invalid or missing, the specified default is returned. For an existing key, the value is converted from string form to the appropriate data type. These functions do not perform any type checking.

Parameters

server

[in] A valid SBServer object.

teamnum

[in] The integer index of the team.

key

[in] The value associated with this key will be returned.

idefault

[in] The value to return if the key is not found.

Remarks

These functions are useful for converting custom keys to a native data type. No type checking is performed, the string value is simply cast to the appropriate data type. If a key is not found, the supplied default is returned.

The SBServer object may be obtained during the SBCallback from ServerBrowserUpdate, or by calling ServerBrowserGetServer. An SBServer object will only exist for servers in the list. IP addresses removed from the server list will not have an SBServer object associated.

Team indexes are determined on a per-game basis. The only requirement is that they match the server's reporting indexes.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE I
SBServerGetTeamIntValue	SBServerGetTeamIntValueA	SBServerGetTea

SBServerGetTeamIntValueW and **SBServerGetTeamIntValueA** are UNICODE and ANSI mapped versions of **SBServerGetTeamIntValue**. The arguments of **SBServerGetTeamIntValueA** are ANSI strings; those of **SBServerGetTeamIntValueW** are wide-character strings.

Section Reference: [Gamespy Server Browsing SDK](#)

See Also: [ServerBrowserNew](#), [ServerBrowserUpdate](#), [ServerBrowserGetServer](#)

SBServerGetTeamStringValue

Returns the value associated with the specified teams' key. This value is returned as the appropriate type; Float, int or string.

```
const gsi_char * SBServerGetTeamStringValue(  
    SBServer server,  
    int teamnum,  
    const gsi_char * key,  
    const gsi_char * sdefault );
```

Routine	Required Header	Distribution
SBServerGetTeamStringValue	<sb_serverbrowsing.h>	SDKZIP

Return Value

If the key is invalid or missing, the specified default is returned. For an existing key, the value is converted from string form to the appropriate data type. These functions do not perform any type checking.

Parameters

server

[in] A valid SBServer object.

teamnum

[in] The integer index of the team.

key

[in] The value associated with this key will be returned.

sdefault

[in] The value to return if the key is not found.

Remarks

These functions are useful for converting custom keys to a native data type. No type checking is performed, the string value is simply cast to the appropriate data type. If a key is not found, the supplied default is returned.

The SBServer object may be obtained during the SBCallback from ServerBrowserUpdate, or by calling ServerBrowserGetServer. An SBServer object will only exist for servers in the list. IP addresses removed from the server list will not have an SBServer object associated.

Team indexes are determined on a per-game basis. The only requirement is that they match the server's reporting indexes.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNIC
SBServerGetTeamStringValue	SBServerGetTeamStringValueA	SBServer

SBServerGetTeamStringValueW and **SBServerGetTeamStringValueA** are UNICODE and ANSI mapped versions of **SBServerGetTeamStringValue**. The arguments of **SBServerGetTeamStringValueA** are ANSI strings; those of **SBServerGetTeamStringValueW** are wide-character strings.

Section Reference: [Gamespy Server Browsing SDK](#)

See Also: [ServerBrowserNew](#), [ServerBrowserUpdate](#), [ServerBrowserGetServer](#)

SBServerHasBasicKeys

Determine if basic information is available for the specified server.

```
SBBool SBServerHasBasicKeys(  
    SBServer server );
```

Routine	Required Header	Distribution
SBServerHasBasicKeys	<sb_serverbrowsing.h>	SDKZIP

Return Value

SBTrue if available; otherwise SBFalse.

Parameters

server

[in] A valid SBServer object.

Remarks

The **SBServerHasBasicKeys** function is used to determine if basic server information is available for the server. Information may not be available if a server query is still pending.

Section Reference: [Gamespy Server Browsing SDK](#)

See Also: [ServerBrowserNew](#), [ServerBrowserUpdate](#), [ServerBrowserGetServer](#)

SBServerHasFullKeys

Determine if full information is available for the specified server.

```
SBBool SBServerHasFullKeys(  
    SBServer server );
```

Routine	Required Header	Distribution
SBServerHasFullKeys	<sb_serverbrowsing.h>	SDKZIP

Return Value

SBTrue if available; otherwise SBFalse.

Parameters

server

[in] A valid SBServer object.

Remarks

The **SBServerHasFullKeys** function is used to determine if full server information is available for the server. Information may not be available if a server query is still pending.

Section Reference: [Gamespy Server Browsing SDK](#)

See Also: [ServerBrowserNew](#), [ServerBrowserUpdate](#), [ServerBrowserGetServer](#)

SBServerHasPrivateAddress

Tests to see if a private address is available for the server.

```
SBBool SBServerHasPrivateAddress(  
    SBServer server );
```

Routine	Required Header	Distribution
SBServerHasPrivateAddress	<sb_serverbrowsing.h>	SDKZIP

Return Value

Returns `SBTrue` if the server has a private address; otherwise `SBFalse`.

Parameters

server

[in] A valid SBServer object.

Section Reference: [Gamespy Server Browsing SDK](#)

See Also: [ServerBrowserNew](#), [ServerBrowserUpdate](#),
[ServerBrowserGetServer](#)

SBServerHasValidPing

Determines if a server has a valid ping value (otherwise the ping will be 0).

```
SBBool SBServerHasValidPing(  
    SBServer server );
```

Routine	Required Header	Distribution
SBServerHasValidPing	<sb_serverbrowsing.h>	SDKZIP

Return Value

SBTrue if the server has a valid ping value, otherwise SBFalse.

Parameters

server

[in] A valid SBServer object.

Section Reference: [Gamespy Server Browsing SDK](#)

ServerBrowserAuxUpdateIP

Queries key/values from a single server.

SBError ServerBrowserAuxUpdateIP(

```
    ServerBrowser sb,  
    const gsi_char * ip,  
    unsigned short port,  
    SBBool viaMaster,  
    SBBool async,  
    SBBool fullUpdate );
```

Routine	Required Header	Distribution
ServerBrowserAuxUpdateIP	<sb_serverbrowsing.h>	SDKZIP

Return Value

This function returns `sbe_noerror` for success. On an error condition, this function will return an `SBError` code. If the `async` option is `SBTrue`, the status condition will be reported to the `SBCallback`.

Parameters

sb

[in] ServerBrowser object returned from ServerBrowserNew.

ip

[in] Address string of the game server.

port

[in] Query port of the game server, in network byte order.

viaMaster

[in] Set to SBTrue to retrieve cached values from the master server.

async

[in] Set to SBTrue to run in non-blocking mode.

fullUpdate

[in] Set to SBTrue to retrieve the full set of key/values from the server.

Remarks

The **ServerBrowserAuxUpdateIP** function is used to retrieve information about a specific server. Information returned is in the form of key/value pairs and may be accessed through the standard SBServer object accessors.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE
ServerBrowserAuxUpdateIP	ServerBrowserAuxUpdateIPA	ServerBrowse

ServerBrowserAuxUpdateIPW and **ServerBrowserAuxUpdateIPA** are UNICODE and ANSI mapped versions of **ServerBrowserAuxUpdateIP**. The arguments of **ServerBrowserAuxUpdateIPA** are ANSI strings; those of **ServerBrowserAuxUpdateIPW** are wide-character strings.

Section Reference: [Gamespy Server Browsing SDK](#)

See Also: [ServerBrowserUpdate](#), [ServerBrowserLANUpdate](#), [ServerBrowserAuxUpdateServer](#)

ServerBrowserAuxUpdateServer

Query key/values from a single server that has already been added to the internal list.

```
SBEError ServerBrowserAuxUpdateServer(  
    ServerBrowser sb,  
    SBServer server,  
    SBBool async,  
    SBBool fullUpdate );
```

Routine	Required Header	Distribution
ServerBrowserAuxUpdateServer	<sb_serverbrowsing.h>	SDKZIP

Return Value

This function returns `sbe_noerror` for success. On an error condition, this function will return an `SBError` code. If the `async` option is `SBTrue`, the status condition will be reported to the `SBCallback`.

Parameters

sb

[in] ServerBrowser object returned from ServerBrowserNew.

server

[in] SBServer object for the server to update. (usually obtained from SBCallback)

async

[in] Set to SBTrue to run in non-blocking mode.

fullUpdate

[in] Set to SBTrue to retrieve the full set of key/values from the server.

Remarks

The **ServerBrowserAuxUpdateServer** function is used to retrieve information about a specific server. Information returned is in the form of key/value pairs and may be accessed through the standard SBServer object accessors.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_
ServerBrowserAuxUpdateServer	ServerBrowserAuxUpdateServerA	Server

ServerBrowserAuxUpdateServerW and **ServerBrowserAuxUpdateServerA** are UNICODE and ANSI mapped versions of **ServerBrowserAuxUpdateServer**. The arguments of **ServerBrowserAuxUpdateServerA** are ANSI strings; those of **ServerBrowserAuxUpdateServerW** are wide-character strings.

Section Reference: [Gamespy Server Browsing SDK](#)

See Also: [ServerBrowserUpdate](#), [ServerBrowserLANUpdate](#), [ServerBrowserAuxUpdateIP](#)

ServerBrowserClear

Clear the current server list.

```
void ServerBrowserClear(  
    ServerBrowser sb );
```

Routine	Required Header	Distribution
ServerBrowserClear	<sb_serverbrowsing.h>	SDKZIP

Parameters

sb

[in] ServerBrowser object initialized with ServerBrowserNew.

Remarks

The **ServerBrowserClear** function empties the current list of servers in preparation for a **ServerBrowserUpdate** or other list populating call.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
ServerBrowserClear	ServerBrowserClearA	ServerBrowserClearW

ServerBrowserClearW and **ServerBrowserClearA** are UNICODE and ANSI mapped versions of **ServerBrowserClear**. The arguments of **ServerBrowserClearA** are ANSI strings; those of **ServerBrowserClearW** are wide-character strings.

Section Reference: [Gamespy Server Browsing SDK](#)

See Also: [ServerBrowserNew](#), [ServerBrowserUpdate](#), [ServerBrowserFree](#)

ServerBrowserConnectToServer

Connects to a game server.

```
SBError ServerBrowserConnectToServer(  
    ServerBrowser sb,  
    SBServer server,  
    SBConnectToServerCallback callback );
```

Routine	Required Header	Distribution
ServerBrowserConnectToServer	<sb_serverbrowsing.h>	SDKZIP

Return Value

This function returns `sbe_noerror` for success. On an error condition, this function will return an `SBEError` code. If there is an error, the callback will not be called.

Parameters

sb

[in] ServerBrowser object returned from ServerBrowserNew.

server

[in] SBServer object for the server to connect to.

callback

[in] The callback to call when the attempt completes.

Remarks

Connects to a game server, internally using Nat Negotiation if necessary. The callback will be called when the connection attempt completes. If the attempt is successful, the server will have its `qr2_clientconnectedcallback_t` called.

Section Reference: [Gamespy Server Browsing SDK](#)

See Also: [QR2\qr2_clientconnectedcallback_t](#),
[SBConnectToServerCallback](#)

ServerBrowserCount

Retrieves the current list of games from the GameSpy master server.

```
int ServerBrowserCount(  
    ServerBrowser sb );
```

Routine	Required Header	Distribution
ServerBrowserCount	<sb_serverbrowsing.h>	SDKZIP

Return Value

Returns the number of servers in the current list. The index is zero based when referencing.

Parameters

sb

[in] ServerBrowser object initialized with ServerBrowserNew.

Remarks

The **ServerBrowserCount** function returns the number of servers in the current list. This may be a combination of servers returned by **ServerBrowserUpdate** and servers added manually by **ServerBrowserAuxUpdateIP**. Please note that index functions such as **ServerBrowserGetServer** use a zero based index. The actual valid indexes are 0 to **ServerBrowserCount()-1**.

Section Reference: [Gamespy Server Browsing SDK](#)

See Also: [ServerBrowserNew](#), [ServerBrowserUpdate](#), [ServerBrowserGetServer](#)

ServerBrowserDisconnect

Disconnect from the GameSpy master server.

```
void ServerBrowserDisconnect(  
    ServerBrowser sb );
```

Routine	Required Header	Distribution
ServerBrowserDisconnect	<sb_serverbrowsing.h>	SDKZIP

Parameters

sb

[in] ServerBrowser object initialized with ServerBrowserNew.

Remarks

The **ServerBrowserDisconnect** function disconnects a maintained connection to the GameSpy master server. This is only necessary when explicitly maintaining a connection to the backend. This should only be done after careful consideration.

Section Reference: [Gamespy Server Browsing SDK](#)

See Also: [ServerBrowserNew](#), [ServerBrowserUpdate](#)

ServerBrowserErrorDesc

Returns a human readable string for the specified SError.

```
const gsi_char * ServerBrowserErrorDesc(  
    ServerBrowser sb,  
    SError error );
```

Routine	Required Header	Distribution
ServerBrowserErrorDesc	<sb_serverbrowsing.h>	SDKZIP

Return Value

For a valid SError, this function will return a human readable description. Otherwise this function returns an empty string.

Parameters

sb

[in] ServerBrowser object initialized with ServerBrowserNew.

error

[in] A valid SBEError code.

Remarks

The **ServerBrowserErrorDesc** function is usefull for displaying error information to a user that might not understand SError codes. These descriptions are in english. For localization purposes, you will need to provide your own translation functions.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defi
ServerBrowserErrorDesc	ServerBrowserErrorDescA	ServerBrowserError

ServerBrowserErrorDescW and **ServerBrowserErrorDescA** are UNICODE and ANSI mapped versions of **ServerBrowserErrorDesc**. The arguments of **ServerBrowserErrorDescA** are ANSI strings; those of **ServerBrowserErrorDescW** are wide-character strings.

Section Reference: [Gamespy Server Browsing SDK](#)

See Also: [ServerBrowserNew](#), [ServerBrowserListQueryError](#)

ServerBrowserFree

Frees memory allocated by the ServerBrowser SDK. Terminates any pending queries.

```
void ServerBrowserFree(  
    ServerBrowser sb );
```

Routine	Required Header	Distribution
ServerBrowserFree	<sb_serverbrowsing.h>	SDKZIP

Parameters

sb

[in] A ServerBrowser interface previously allocated with ServerBrowserNew.

Remarks

The **ServerBrowserFree** function frees any allocated memory associated with the SDK as well as terminates any pending queries. This function must be called once for every call to **ServerBrowserNew** to ensure proper cleanup of the ServerBrowsing SDK.

Example

```
/* SERVERBROWSERFREE.C: This program uses ServerBrowserNew * to init
#include <sb_serverbrowsing.h>

void main( void )
{
    ServerBrowser aServerBrowser = ServerBrowserNew("gmttest", "gmtes
    ServerBrowserFree(aServerBrowser);
}
```

Section Reference: [Gamespy Server Browsing SDK](#)

See Also: [ServerBrowserNew](#)

ServerBrowserGetMyPublicIP

Returns the local client's external (firewall) address.

```
char * ServerBrowserGetMyPublicIP(  
    ServerBrowser sb );
```

Routine	Required Header	Distribution
ServerBrowserGetMyPublicIP	<sb_serverbrowsing.h>	SDKZIP

Return Value

The local clients external (firewall) address. This may be returned as a string or integer address.

Parameters

sb

[in] ServerBrowser object initialized with ServerBrowserNew

Remarks

The **ServerBrowserGetMyPublicIP** and **ServerBrowserGetMyPublicIPAddr** functions return the external address of the local client, as report by the GameSpy Master Server. Because of this, the return value is only valid after a successful call to **ServerBrowserUpdate**. The reason for this is that a client cannot determine their external address without first sending an outgoing packet. It is up to the receiver of that packet to report the public address back to the local client.

Section Reference: [Gamespy Server Browsing SDK](#)

See Also: [ServerBrowserNew](#), [ServerBrowserUpdate](#)

ServerBrowserGetMyPublicIPAddr

Returns the local client's external (firewall) address.

```
unsigned int ServerBrowserGetMyPublicIPAddr(  
    ServerBrowser sb );
```

Routine	Required Header	Distribution
ServerBrowserGetMyPublicIPAddr	<sb_serverbrowsing.h>	SDKZIP

Return Value

The local clients external (firewall) address. This may be returned as a string or integer address.

Parameters

sb

[in] ServerBrowser object initialized with ServerBrowserNew

Remarks

The `ServerBrowserGetMyPublicIP` and **`ServerBrowserGetMyPublicIPAddr`** functions return the external address of the local client, as report by the GameSpy Master Server. Because of this, the return value is only valid after a successful call to `ServerBrowserUpdate`. The reason for this is that a client cannot determine their external address without first sending an outgoing packet. It is up to the receiver of that packet to report the public address back to the local client.

Section Reference: [Gamespy Server Browsing SDK](#)

See Also: [ServerBrowserNew](#), [ServerBrowserUpdate](#)

ServerBrowserGetServer

Returns the SBServer object at the specified index.

```
SBServer ServerBrowserGetServer(  
    ServerBrowser sb,  
    int index );
```

Routine	Required Header	Distribution
ServerBrowserGetServer	<sb_serverbrowsing.h>	SDKZIP

Return Value

Returns the SBServer at the specified array index. If index is greater than the bounds of the array, NULL is returned.

Parameters

sb

[in] ServerBrowser object initialized with ServerBrowserNew.

index

[in] The array index.

Remarks

Use `ServerBrowserCount` to retrieve the current number of servers in the array. This index is zero based, so a list containing 5 servers will have the valid indexes 0 through 4. This list is usually populated using one of the list retrieval methods such as `ServerBrowserUpdate`.

Section Reference: [Gamespy Server Browsing SDK](#)

See Also: [ServerBrowserNew](#), [ServerBrowserUpdate](#)

ServerBrowserGetServerByIP

Returns the SBServer with the specified IP.

```
SBServer ServerBrowserGetServerByIP(  
    ServerBrowser sb,  
    const gsi_char * ip,  
    unsigned short port );
```

Routine	Required Header	Distribution
ServerBrowserGetServerByIP	<sb_serverbrowsing.h>	SDKZIP

Return Value

Returns the Server if found, otherwise NULL;

Parameters

sb

[in] ServerBrowser object initialized with ServerBrowserNew.

ip

[in] The dotted IP address of the server e.g. "1.2.3.4"

port

[in] The query port of the server, in network byte order.

Section Reference: [Gamespy Server Browsing SDK](#)

ServerBrowserHalt

Stop an update in progress. .

```
void ServerBrowserHalt(  
    ServerBrowser sb );
```

Routine	Required Header	Distribution
ServerBrowserHalt	<sb_serverbrowsing.h>	SDKZIP

Parameters

sb

[in] ServerBrowser object initialized with ServerBrowserNew.

Remarks

The **ServerBrowserHalt** function will stop an update in progress. This is often tied to a "cancel" button presented to the user on the server list screen. Clears any servers queued to be queried, and disconnects from the master server.

Section Reference: [Gamespy Server Browsing SDK](#)

See Also: [ServerBrowserNew](#), [ServerBrowserUpdate](#)

ServerBrowserLANSetLocalAddr

Sets the network adapter to use for LAN broadcasts (optional).

```
void ServerBrowserLANSetLocalAddr(  
    ServerBrowser sb,  
    const char * theAddr );
```

Routine	Required Header	Distribution
ServerBrowserLANSetLocalAddr	<sb_serverbrowsing.h>	SDKZIP

Parameters

sb

[in] ServerBrowser object initialized with ServerBrowserNew.

theAddr

[in] The address to use.

Section Reference: [Gamespy Server Browsing SDK](#)

ServerBrowserLANUpdate

Retrieves the current list of games broadcasting on the local network.

```
SBError ServerBrowserLANUpdate(  
    ServerBrowser sb,  
    SBBool async,  
    unsigned short startSearchPort,  
    unsigned short endSearchPort );
```

Routine	Required Header	Distribution
ServerBrowserLANUpdate	<sb_serverbrowsing.h>	SDKZIP

Return Value

If an error occurs, a valid SBEError error code is returned. Otherwise, sbe_noerror is returned.

Parameters

sb

[in] ServerBrowser object initialized with ServerBrowserNew.

async

[in] When set to SBTrue this function will run in non-blocking mode.

startSearchPort

[in] The lowest port the SDK will listen to broadcasts from, in network byte order.

endSearchPort

[in] The highest port the SDK will listen to broadcasts from, in network byte order.

Remarks

The **ServerBrowserLANUpdate** function listens for broadcast packets on the local network. Servers that are broadcasting within the specified port range will be detected. As each server broadcast is received, one corresponding call to the `SBCallbackfunction` will be made with the status `sbc_serveradded`.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE D
ServerBrowserLANUpdate	ServerBrowserLANUpdateA	ServerBrowserLA

ServerBrowserLANUpdateW and **ServerBrowserLANUpdateA** are UNICODE and ANSI mapped versions of **ServerBrowserLANUpdate**. The arguments of **ServerBrowserLANUpdateA** are ANSI strings; those of **ServerBrowserLANUpdateW** are wide-character strings.

Section Reference: [Gamespy Server Browsing SDK](#)

See Also: [ServerBrowserNew](#), [ServerBrowserUpdate](#)

ServerBrowserLimitUpdate

Retrieves the current limited list of games from the GameSpy master server. Useful for low-memory systems.

```
SBError ServerBrowserLimitUpdate(  
    ServerBrowser sb,  
    SBBool async,  
    SBBool disconnectOnComplete,  
    const unsigned char * basicFields,  
    int numBasicFields,  
    const gsi_char * serverFilter,  
    int maxServers );
```

Routine	Required Header	Distribution
ServerBrowserLimitUpdate	<sb_serverbrowsing.h>	SDKZIP

Return Value

If an error occurs, a valid SBEError error code is returned. Otherwise, sbe_noerror is returned.

Parameters

sb

[in] ServerBrowser object initialized with ServerBrowserNew.

async

[in] When set to SBTrue this function will run in non-blocking mode.

disconnectOnComplete

[in] When set to SBTrue this function will terminate the connection with the GameSpy master after the download is complete.

basicFields

[in] A byte array of basic field identifiers to retrieve for each server.
See remarks.

numBasicFields

[in] The number of valid fields in the basicFields array.

serverFilter

[in] SQL like string used to remove unwanted servers from the downloaded list.

maxServers

[in] Maximum number of servers to be returned

Remarks

The **ServerBrowserLimitUpdate** function retrieves a limited set of the servers registered with the GameSpy master server. This is most useful for low memory systems such as the PS2 which may not be capable of loading an entire server list.

Identical to `ServerBrowserUpdate`, except that the number of servers returned can be limited.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE I
ServerBrowserLimitUpdate	ServerBrowserLimitUpdateA	ServerBrowserLi

ServerBrowserLimitUpdateW and **ServerBrowserLimitUpdateA** are UNICODE and ANSI mapped versions of **ServerBrowserLimitUpdate**. The arguments of **ServerBrowserLimitUpdateA** are ANSI strings; those of **ServerBrowserLimitUpdateW** are wide-character strings.

Section Reference: [Gamespy Server Browsing SDK](#)

See Also: [ServerBrowserNew](#), [ServerBrowserUpdate](#)

ServerBrowserListQueryError

Returns the ServerList error string, if any.

```
const gsi_char * ServerBrowserListQueryError(  
    ServerBrowser sb );
```

Routine	Required Header	Distribution
ServerBrowserListQueryError	<sb_serverbrowsing.h>	SDKZIP

Return Value

If a list error has occurred, a string description of the error is returned. Otherwise, an empty string "" is returned.

Parameters

sb

[in] ServerBrowser object initialized with ServerBrowserNew.

Remarks

The **ServerBrowserListQueryError** function returns the last string error reported by the server. For localization purposes, you may safely assume that this string will not change, and test for it as you would a numeric error code.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE
ServerBrowserListQueryError	ServerBrowserListQueryErrorA	ServerBrowserListQueryErrorW

ServerBrowserListQueryErrorW and **ServerBrowserListQueryErrorA** are UNICODE and ANSI mapped versions of **ServerBrowserListQueryError**. The arguments of **ServerBrowserListQueryErrorA** are ANSI strings; those of **ServerBrowserListQueryErrorW** are wide-character strings.

Section Reference: [Gamespy Server Browsing SDK](#)

See Also: [ServerBrowserNew](#)

ServerBrowserNew

Initialize the ServerBrowser SDK.

```
ServerBrowser ServerBrowserNew(  
    const gsi_char * queryForGamename,  
    const gsi_char * queryFromGamename,  
    const gsi_char * queryFromKey,  
    int queryFromVersion,  
    int maxConcUpdates,  
    int queryVersion,  
    SBBool lanBrowse,  
    ServerBrowserCallback callback,  
    void * instance );
```

Routine	Required Header	Distribution
ServerBrowserNew	<sb_serverbrowsing.h>	SDKZIP

Return Value

This function returns the initialized ServerBrowser interface. No return value is reserved to indicate an error.

Parameters

queryForGamename

[in] Servers returned will be for this Gamename.

queryFromGamename

[in] Your assigned GameName.

queryFromKey

[in] Secret key that corresponds to the queryFromGamename.

queryFromVersion

[in] Set to zero unless directed otherwise by GameSpy.

maxConcUpdates

[in] The maximum number of queries the ServerBrowsing SDK will send out at one time.

queryVersion

[in] The QueryReporting protocol used by the server. Should be QVERSION_GOA or QVERSION_QR2. See remarks.

lanBrowse

[in] The switch to turn on only LAN browsing

callback

[in] Function to be called when the operation completes.

instance

[in] Pointer to user data. This is optional and will be passed unmodified to the callback function.

Remarks

The **ServerBrowserNew** function initializes the ServerBrowsing SDK. Developers should then use `ServerBrowserUpdate` or `ServerBrowserLANUpdate` to begin retrieving the list of registered game servers.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
ServerBrowserNew	ServerBrowserNewA	ServerBrowserNewW

ServerBrowserNewW and **ServerBrowserNewA** are UNICODE and ANSI mapped versions of **ServerBrowserNew**. The arguments of **ServerBrowserNewA** are ANSI strings; those of **ServerBrowserNewW** are wide-character strings.

Example

(In this particular file, we should refer them to the ServerBrowser

Section Reference: [Gamespy Server Browsing SDK](#)

See Also: [ServerBrowserFree](#), [ServerBrowserUpdate](#),
[ServerBrowserLANUpdate](#)

ServerBrowserPendingQueryCount

Retrieves the number of servers with outstanding queries. Use this to check progress while asynchronously updating the server list.

```
int ServerBrowserPendingQueryCount(  
    ServerBrowser sb );
```

Routine	Required Header	Distribution
ServerBrowserPendingQueryCount	<sb_serverbrowsing.h>	SDKZIP

Return Value

Returns the number of servers that have not yet been queried.

Parameters

sb

[in] ServerBrowser object initialized with ServerBrowserNew.

Remarks

The **ServerBrowserPendingQueryCount** function is most usefull when updating a large list of servers. Use this function to display progress information to the user. For example "1048/2063 servers updated", or as a progress bar graphic.

Section Reference: [Gamespy Server Browsing SDK](#)

See Also: [ServerBrowserNew](#), [ServerBrowserUpdate](#)

ServerBrowserRemoveIP

Removes a server from the local list.

```
void ServerBrowserRemoveIP(  
    ServerBrowser sb,  
    const gsi_char * ip,  
    unsigned short port );
```

Routine	Required Header	Distribution
ServerBrowserRemoveIP	<sb_serverbrowsing.h>	SDKZIP

Parameters

sb

[in] ServerBrowser object initialized with ServerBrowserNew.

ip

[in] The address of the server to remove.

port

[in] The port of the server to remove, in network byte order.

Remarks

The **ServerBrowserRemoveIP** function removes a single SBServer from the local list. This does not affect the backend or remote users. Please refer to the QR2 SDK for removing a registered server from the backend list.

Section Reference: [Gamespy Server Browsing SDK](#)

See Also: [ServerBrowserNew](#), [ServerBrowserUpdate](#), [ServerBrowserRemoveServer](#)

ServerBrowserRemoveServer

Removes a server from the local list.

```
void ServerBrowserRemoveServer(  
    ServerBrowser sb,  
    SBServer server );
```

Routine	Required Header	Distribution
ServerBrowserRemoveServer	<sb_serverbrowsing.h>	SDKZIP

Parameters

sb

[in] ServerBrowser object initialized with ServerBrowserNew.

server

[in] The server to remove.

Remarks

The **ServerBrowserRemoveServer** function removes a single SBServer from the local list. This does not affect the backend or remote users. Please refer to the QR2 SDK for removing a registered server from the backend list.

Section Reference: [Gamespy Server Browsing SDK](#)

See Also: [ServerBrowserNew](#), [ServerBrowserUpdate](#), [ServerBrowserRemoveIP](#)

ServerBrowserSendMessageToServer

Sends a game specific message to the specified IP/port. This message is routed through the master server.

```
SBError ServerBrowserSendMessageToServer(  
    ServerBrowser sb,  
    const gsi_char * ip,  
    unsigned short port,  
    const char * data,  
    int len );
```

Routine	Required Header	Distributi
ServerBrowserSendMessageToServer	<sb_serverbrowsing.h>	SDKZIP

Return Value

If an error occurs, a valid SBEError error code is returned. Otherwise, sbe_noerror is returned.

Parameters

sb

[in] ServerBrowser object initialized with ServerBrowserNew.

ip

[in] Address of the server in string form. "xxx.xxx.xxx.xxx"

port

[in] The query port of the server to send the message to, in network byte order.

data

[in] The raw data buffer.

len

[in] The length of the data buffer.

Remarks

The **ServerBrowserSendMessageToServer** function can be used to relay a raw data buffer to a server behind a firewall. The raw buffer is sent through the backend since direct communication with the server is not always possible. The buffer is sent in raw form to the server's query port and does not contain any header information. This message is most usefull in a shared socket QR2 implementation.

Unicode Mappings

Routine	GSI_UNICODE Not Defined
ServerBrowserSendMessageToServer	ServerBrowserSendMessageToSe

ServerBrowserSendMessageToServerW and **ServerBrowserSendMessageToServerA** are UNICODE and ANSI mapped versions of **ServerBrowserSendMessageToServer**. The arguments of **ServerBrowserSendMessageToServerA** are ANSI strings; those of **ServerBrowserSendMessageToServerW** are wide-character strings.

Section Reference: [Gamespy Server Browsing SDK](#)

See Also: [ServerBrowserNew](#), [ServerBrowserUpdate](#), [ServerBrowserSendNatNegotiateCookieToServer](#)

ServerBrowserSendNatNegotiateCookieToServer

Sends a nat negotiation cookie to the server. The cookie is sent via the master server.

```
SBError ServerBrowserSendNatNegotiateCookieToServer(  
    ServerBrowser sb,  
    const gsi_char * ip,  
    unsigned short port,  
    int cookie );
```

Routine	Required Header
ServerBrowserSendNatNegotiateCookieToServer	<sb_serverbrowsing.h>

Return Value

If an error occurs, a valid SBEError error code is returned. Otherwise, sbe_noerror is returned.

Parameters

sb

[in] ServerBrowser object initialized with ServerBrowserNew.

ip

[in] Address of the server in string form. "xxx.xxx.xxx.xxx"

port

[in] The query port of the server to relay the NatNeg cookie to, in network byte order.

cookie

[in] An integer cookie value. See remarks.

Remarks

The **ServerBrowserSendNatNegotiateCookieToServer** function can be used to relay a NatNegotiation cookie value to a server behind a firewall. This cookie is sent through the backend since direct communication with the server is not always possible. This cookie may then be used to initiate a nat negotiation attempt. Please refer to the NatNegotiation SDK documentation for more info.

Unicode Mappings

Routine	GSI_UNICODE Not De
ServerBrowserSendNatNegotiateCookieToServer	ServerBrowserSendNa

ServerBrowserSendNatNegotiateCookieToServerW and **ServerBrowserSendNatNegotiateCookieToServerA** are UNICODE and ANSI mapped versions of **ServerBrowserSendNatNegotiateCookieToServer**. The arguments of **ServerBrowserSendNatNegotiateCookieToServerA** are ANSI strings; those of **ServerBrowserSendNatNegotiateCookieToServerW** are wide-character strings.

Section Reference: [Gamespy Server Browsing SDK](#)

See Also: [ServerBrowserNew](#), [ServerBrowserLANUpdate](#)

ServerBrowserSort

Sort the current list of servers.

```
void ServerBrowserSort(  
    ServerBrowser sb,  
    SBBool ascending,  
    gsi_char * sortkey,  
    SBCmpareMode comparemode );
```

Routine	Required Header	Distribution
ServerBrowserSort	<sb_serverbrowsing.h>	SDKZIP

Parameters

sb

[in] ServerBrowser object initialized with ServerBrowserNew.

ascending

[in] When set to SBTrue this function will sort in ascending order. (a-b-c order, not c-b-a)

sortkey

[in] The "key" of the key/value pair to sort by.

comparemode

[in] Specifies the data type of the sortkey. See remarks.

Remarks

The **ServerBrowserSort** function will return an ordered list of servers, sorted by the specified sortkey. Sorting may be in ascending or descending order and various data-types are supported. SBCCompareMode may be one of the following values:

sbcm_int: Uses integer comparison. "1,2,3,12,15,20"

sbcm_float: Similar to above but considers decimal values.
"1.1,1.2,2.1,3.0"

sbcm_strcase: Uses case sensitive string comparison. Uses strcmp.

sbcm_stricase: Case in-sensitive string comparison. Uses _stricmp or equivilent.

Please note that calling this function repeatedly for a large server list may impact performance. This is due to the standard qsort algorithm being inefficient when sorting an already ordered list. This is rarely a cause for concern, but certain optimizations may be made if performance is noticeably impacted.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
ServerBrowserSort	ServerBrowserSortA	ServerBrowserSortW

ServerBrowserSortW and **ServerBrowserSortA** are UNICODE and ANSI mapped versions of **ServerBrowserSort**. The arguments of **ServerBrowserSortA** are ANSI strings; those of **ServerBrowserSortW** are wide-character strings.

Section Reference: [Gamespy Server Browsing SDK](#)

See Also: [ServerBrowserUpdate](#), [ServerBrowserThink](#)

ServerBrowserState

Gets current state of the Server Browser object.

```
SBState ServerBrowserState(  
    ServerBrowser sb );
```

Routine	Required Header	Distribution
ServerBrowserState	<sb_serverbrowsing.h>	SDKZIP

Return Value

Returns the current state.

Parameters

sb

[in] ServerBrowser object initialized with ServerBrowserNew.

Remarks

Descriptions of the possible state values can be find in the main header file.

Section Reference: [Gamespy Server Browsing SDK](#)

ServerBrowserThink

Allows ServerBrowsingSDK to continue internal processing including processing query replies.

```
SBError ServerBrowserThink(  
    ServerBrowser sb );
```

Routine	Required Header	Distribution
ServerBrowserThink	<sb_serverbrowsing.h>	SDKZIP

Return Value

If an error occurs, a valid SBEError error code is returned. Otherwise, sbe_noerror is returned.

Parameters

sb

[in] ServerBrowser object initialized with ServerBrowserNew.

Remarks

The **ServerBrowserThink** function is required for the SDK to process incoming data. Because of the single threaded design of the GameSpy SDKs, all data is processed during this call, and processing is paused when this call is complete. When updating server lists, this function should be called as frequently as possible to reduce the latency associated with server response times. If this function is not called often enough, server pings may be inflated due to processing delays.

Section Reference: [Gamespy Server Browsing SDK](#)

See Also: [ServerBrowserNew](#)

ServerBrowserUpdate

Retrieves the current list of games from the GameSpy master server.

```
SBError ServerBrowserUpdate(  
    ServerBrowser sb,  
    SBBool async,  
    SBBool disconnectOnComplete,  
    const unsigned char * basicFields,  
    int numBasicFields,  
    const gsi_char * serverFilter );
```

Routine	Required Header	Distribution
ServerBrowserUpdate	<sb_serverbrowsing.h>	SDKZIP

Return Value

If an error occurs, a valid SBEError error code is returned. Otherwise, sbe_noerror is returned.

Parameters

sb

[in] ServerBrowser object initialized with ServerBrowserNew.

async

[in] When set to SBTrue this function will run in non-blocking mode.

disconnectOnComplete

[in] When set to SBTrue this function will terminate the connection with the GameSpy master after the download is complete.

basicFields

[in] A byte array of basic field identifiers to retrieve for each server.
See remarks.

numBasicFields

[in] The number of valid fields in the basicFields array.

serverFilter

[in] SQL like string used to remove unwanted servers from the downloaded list.

Remarks

The **ServerBrowserUpdate** function retrieves the current list of servers registered with the GameSpy master server. As each server entry is received, one corresponding call to the SBCallback function will be made with the status `sbc_serveradded`.

Unicode Mappings

Routine	GSI_UNICODE Not Defined	GSI_UNICODE Defined
ServerBrowserUpdate	ServerBrowserUpdateA	ServerBrowserUpdateW

ServerBrowserUpdateW and **ServerBrowserUpdateA** are UNICODE and ANSI mapped versions of **ServerBrowserUpdate**. The arguments of **ServerBrowserUpdateA** are ANSI strings; those of **ServerBrowserUpdateW** are wide-character strings.

Section Reference: [Gamespy Server Browsing SDK](#)

See Also: [ServerBrowserNew](#), [ServerBrowserLANUpdate](#)

Server Browsing SDK Callbacks

[SBConnectToServerCallback](#)

The callback provided to ServerBrowserConnectToServer. Gets called when the state of the connect attempt changes.

[SBServerKeyEnumFn](#)

Callback function used for enumerating the keys/values for a server

[ServerBrowserCallback](#)

The callback provided to ServerBrowserNew. Gets called as the Server Browser updates the server list.

SBCConnectToServerCallback

The callback provided to ServerBrowserConnectToServer. Gets called when the state of the connect attempt changes.

```
typedef void (*SBCConnectToServerCallback)(  
    ServerBrowser sb,  
    SBCConnectToServerState state,  
    SOCKET gameSocket,  
    struct sockaddr_in * remoteaddr,  
    void * instance );
```

Routine	Required Header	Distribution
SBCConnectToServerCallback	<sb_serverbrowsing.h>	SDKZIP

Parameters

sb

[in] The ServerBrowser object the callback is referring to.

state

[in] The state of the connect attempt.

gameSocket

[in] A UDP socket, ready for use to communicate with the server.

remoteaddr

[in] The address of the server.

instance

[in] User provided data.

Remarks

"instance" is any game-specific data you want passed to the callback function. For example, you can pass a structure pointer or object pointer for use within the CallBack. If you can access any needed data within the CallBack already, then you can just pass NULL for "instance".

Section Reference: [Gamespy Server Browsing SDK](#)

See Also: [ServerBrowserConnectToServer](#)

SBServerKeyEnumFn

Callback function used for enumerating the keys/values for a server.

```
typedef void (*SBServerKeyEnumFn)(  
    gsi_char * key,  
    gsi_char * value,  
    void * instance );
```

Routine	Required Header	Distribution
SBServerKeyEnumFn	<sb_serverbrowsing.h>	SDKZIP

Parameters

key

[in] The enumerated key.

value

[in] The enumerated value.

instance

[in] User provided data.

Section Reference: [Gamespy Server Browsing SDK](#)

See Also: [SBServerEnumKeys](#)

ServerBrowserCallback

The callback provided to ServerBrowserNew. Gets called as the Server Browser updates the server list.

```
typedef void (*ServerBrowserCallback)(  
    ServerBrowser sb,  
    SBCallbackReason reason,  
    SBServer server,  
    void * instance );
```

Routine	Required Header	Distribution
ServerBrowserCallback	<sb_serverbrowsing.h>	SDKZIP

Parameters

sb

[in] The ServerBrowser object the callback is referring to.

reason

[in] The reason for being called. See SDK Doc for more info.

server

[in] The server that is being referred to.

instance

[in] User provided data.

Remarks

"instance" is any game-specific data you want passed to the callback function. For example, you can pass a structure pointer or object pointer for use within the CallBack. If you can access any needed data within the CallBack already, then you can just pass NULL for "instance".

Example

Your callback function should look something like:

```
void SBCallback(ServerBrowser sb, SBCallbackReason reason, SBServer
{
    CMyGame *g = (CMyGame *)instance;

    switch (reason)
    {
    case sbc_serveradded :
        g->ServerView->AddServerToList(server);
        break;
    case sbc_serverupdated :
        g->ServerView->UpdateServerInList(server);
        break;
    case sbc_updatecomplete:
        g->ServerView->SetStatus("Update Complete");
        break;
    case sbc_queryerror:
        g->ServerView->SetStatus("Query Error Occurred:",
        ServerBrowserListQueryError(sb));
        break;
    }
}
```

Example use of the Callback:

```
int CMyGame::OnMultiplayerButtonClicked(...)
{
    m_ServerBrowser = ServerBrowserNew("mygame", "mygame", "123456",
                                        QVER
}
```

Section Reference: [Gamespy Server Browsing SDK](#)

See Also: [ServerBrowserNew](#)

Server Browsing SDK Enumerations

SBBool	Standard Boolean.
SBCallbackReason	Callbacks that can occur during server browsing operations.
SBCompareMode	Comparison types for the ServerBrowserSort function.
SBConnectToServerState	States passed to the SBConnectToServerCallback.
SBError	Error codes that can be returned from Server Browsing functions.
SBState	States the ServerBrowser object can be in.

SBBool

Standard Boolean.

```
typedef enum  
{  
    SBFalse,  
    SBTrue  
} SBBool;
```

Constants

SBFalse
False.

SBTrue
True.

Section Reference: [Gamespy Server Browsing SDK](#)

SBCallbackReason

Callbacks that can occur during server browsing operations.

```
typedef enum  
{  
    sbc_serveradded,  
    sbc_serverupdated,  
    sbc_serverupdatefailed,  
    sbc_serverdeleted,  
    sbc_updatecomplete,  
    sbc_queryerror,  
    sbc_serverchallengereceived  
} SBCallbackReason;
```

Constants

sbc_serveradded

A server was added to the list, may just have an IP & port at this point.

sbc_serverupdated

Server information has been updated - either basic or full information is now available about this server.

sbc_serverupdatefailed

An attempt to retrieve information about this server, either directly or from the master, failed.

sbc_serverdeleted

A server was removed from the list.

sbc_updatecomplete

The server query engine is now idle.

sbc_queryerror

The master returned an error string for the provided query.

sbc_serverchallengeceived

Prequery ip verification challenge was received. (Informational, no action required.).

Section Reference: [Gamespy Server Browsing SDK](#)

SBCCompareMode

Comparison types for the ServerBrowserSort function.

```
typedef enum  
{  
    sbcm_int,  
    sbcm_float,  
    sbcm_strcase,  
    sbcm_stricase  
} SBCCompareMode;
```

Constants

sbcm_int

Assume the values are int, and do an integer compare.

sbcm_float

Assume the values are float, and do a float compare.

sbcm_strcase

Assume the values are strings, and do a case-sensitive compare.

sbcm_stricase

Assume the values are strings, and do a case-insensitive compare.

Section Reference: [Gamespy Server Browsing SDK](#)

See Also: [ServerBrowserSort](#)

SBCConnectToServerState

States passed to the SBCConnectToServerCallback.

```
typedef enum  
{  
    sbc_succeeded,  
    sbc_failed  
} SBCConnectToServerState;
```

Constants

sbc_s_succeeded

Connected to server successfully.

sbc_s_failed

Failed to connect to server.

Section Reference: [Gamespy Server Browsing SDK](#)

See Also: [ServerBrowserState](#)

SBEError

Error codes that can be returned from Server Browsing functions.

```
typedef enum  
{  
    sbe_noerror,  
    sbe_socketerror,  
    sbe_dnserror,  
    sbe_connecterror,  
    sbe_dataerror,  
    sbe_allocerror,  
    sbe_paramerror  
} SBEError;
```

Constants

sbe_noerror

No error has occurred.

sbe_socketerror

A socket function has returned an unexpected error.

sbe_dnsererror

DNS lookup of master address failed.

sbe_connecterror

Connection to master server failed.

sbe_dataerror

Invalid data was returned from master server.

sbe_allocerror

Memory allocation failed.

sbe_paramerror

An invalid parameter was passed to a function.

Section Reference: [Gamespy Server Browsing SDK](#)

SBState

States the ServerBrowser object can be in.

```
typedef enum  
{  
    sb_disconnected,  
    sb_listxfer,  
    sb_querying,  
    sb_connected  
} SBState;
```

Constants

sb_disconnected

Idle and not connected to the master server.

sb_listxfer

Downloading list of servers from the master server.

sb_querying

Querying servers.

sb_connected

Idle but still connected to the master server.

Section Reference: [Gamespy Server Browsing SDK](#)

See Also: [ServerBrowserState](#)

Transport 2 SDK

Overview

The GameSpy Transport SDK 2 (GT2) is a library that allows two applications to communicate over the Internet, making use of UDP for both reliable and unreliable messaging. It can be used to write any sort of networked application, including both peer-to-peer and dedicated server games. Someone with little or no networking experience can easily learn GT2, without having to learn all the complexities of Sockets/Winsock, and without having to deal with all the overhead involved in DirectPlay.

GT2 is basic enough to be easily and quickly added to an application, while also being powerful and flexible enough to fit within virtually any networking architecture. And, because GT2 is at a lower level than something like DirectPlay, it is extremely efficient in its use of memory, bandwidth, and processor time. So, GT2 delivers optimal performance, in a simple API, while avoiding the hidden traps involved in low level libraries such as Sockets/Winsock and cutting out the overhead and loss of flexibility that comes with a higher level library such as DirectPlay.

The SDK is written in standard ANSI C and has been tested on Win32, Unix, Mac, and consoles. The library has been designed to be easy to use, fast, and memory efficient (particularly useful on console systems with tight memory requirements). Just include all of the source files in your project, and you can start easily communicating over the Internet.

The SDK also includes five samples. *gt2testc* is a simple ANSI C sample that is good for testing without a graphical interface (e.g., on a console), *gt2test* is a Windows MFC sample that is good for testing all the various features of GT2, *gt2proxy* is a GT2 proxy, *gt2hostmig* shows host migration using GT2 and, optionally, the Query & Reporting SDK, and *gt2action* is a sample game that uses GT2 for it's networking.

The rest of this document presents a simple set of instructions for using GT2. See the reference documentation for more detailed information on each function.

File Manifest

The following files should be included with this package. If any of the files are missing, please [contact devsupport@gamespy.com](mailto:devsupport@gamespy.com).

File	Description
<i>gt2.h</i> here)	GT2 header (all user functions are prototyped here)
<i>gt2Main.c,h</i>	The main entry point for most GT2 functionality
<i>gt2Auth.c,h</i> connection negotiation	This code deals with authentication during connection negotiation
<i>gt2Buffer.c,h</i>	Code that deals with reading/writing from buffers
<i>gt2Callback.c,h</i>	Code for calling callbacks
<i>gt2Connection.c,h</i>	This code manages GTConnection objects
<i>gt2Encode.c,h</i> binary format	Sub-library for encoding/decoding messages in a binary format
<i>gt2Filter.c,h</i>	This code manages filtering
<i>gt2Message.c,h</i>	Code for sending and receiving messages
<i>gt2Socket.c,h</i>	This code manages GTSocket objects
<i>gt2Utility.c,h</i> address functions	Code for various utility functions such as the address functions
<i>nonport.c,h</i>	Platform-specific code
<i>darray.c,h</i>	Code for managing dynamic arrays
<i>hastable.c,h</i>	Code for managing hashtables
<i>/gt2testc/</i>	ANSI-C sample
<i>/gt2test/</i>	Windows MFC sample
<i>/gt2proxy/</i>	GT2 Proxy Sample
<i>/gt2hostmig/</i>	GT2 Host-migration sample
<i>/gt2nat/</i>	GT2 Socket Sharing sample. The GT2Socket's underlying socket is also used to do server reporting through the

GameSpy Query and Reporting SDK

/gt2action/ A real-time multiplayer game that uses GT.

GT2Action requires GLUT, a cross-platform utility library that provides windowing for OpenGL applications. The official GLUT site is

<http://www.opengl.org/developers/documentation/glut.html>. GLUT for Windows is available at <http://www.xmission.com/~nate/glut.html>.

Implementation

Sockets and Connections

There are just two object types used by GT2: [GT2Socket](#) and [GT2Connection](#). A [GT2Socket](#) object ("socket") represents a UDP socket on the local machine, and a [GT2Connection](#) object ("connection") represents a connection, or link, between two [GT2Socket](#)'s (in other words, two applications). For most applications, only one socket needs to be created. All incoming connections can be accepted on the socket, and all outgoing connections can be made using the socket.

Thinking

In order for GT2 to do necessary processing on sockets and connections, the program must allow it to frequently "think". This is done by calling [gt2Think](#), which will process the socket that is passed to it, along with all of that socket's connections. Within [gt2Think](#), GT2 will check for new incoming connections on sockets that are listening for connections, do any negotiating needed for pending connections, check for incoming data on connections, check for closed connections, and send any data buffered for a connection.

Internet Addresses

When creating a socket you can optionally specify the IP/hostname and/or port to use on the local machine. You also must specify the IP and port of the remote system being connected to when initiating a connection. These addresses are specified as strings of the form "[IP | hostname][:port]". In other words, if there is a colon in the string, the part before the colon is the IP/hostname, and the part after the colon is the port. If there is no colon, then the whole string is the IP/hostname. When specifying a local address, both the IP/hostname and port are optional. If the IP is not present, then no local IP will be bound to. If the port is not present, then the system will pick an available port to bind to. The utility functions [gt2AddressToString](#) and [gt2StringToAddress](#) are provided to allow for conversion from a string to an IP and port, and vice

versa.

Byte Order

Because the order of the bytes inside a multi-byte variable (such as an int or a short) can vary from system to system, there is the concept of a "network byte order" and a "host byte order". The host byte order is the byte ordering scheme use on the local machine, and network byte order is the byte ordering scheme that is the standard for networking and, specifically, the Internet. In several GT2 functions and callbacks, an IP address and port number are passed around. Whenever GT2 deals with an IP address, it passes it or expects it in NETWORK byte order, and whenever GT2 deals with a port, it passes it or expects it in HOST byte order.

The reason network byte ordering is used for IP addresses is that this is the standard used by the sockets/Winsock functions, and this ensures that the first byte pointed to is the first number in the dotted IP, the second byte is the second number, and so on. The reason host byte ordering is used for port numbers is that this enables the application to pass port numbers directly between it's interface and GT2, or hardcode in a port number (for example with a define), without having to do any byte order conversions.

Byte ordering must also be taken into consideration when sending multi-byte variables in GT2 messages. The safest way to send an int or a short is to use the GT2 byte ordering functions to convert the numbers to network byte order when sending them, then converting them back to host byte order when they are received. This allows one code path to handle sending data between machines that use either the same or different byte ordering schemes.

AdHoc Support

AdHoc is supported in order to allow developers to write to a common layer for both adhoc and infrastructure modes on a hand held console. Adhoc support is provided through an adhoc specific `gt2CreateSocket` function, [gt2CreateAdHocSocket](#). AdHoc sockets use MAC addresses instead of IP addresses to determining end points. Two

functions [gt2IpToMac](#) and [gt2MacToIP](#) are used to convert back and forth between MAC addresses and IPs. GT2 stores internally a table to convert fully back to 48 bit mac from a 32 bit IP. Always call [gt2MacToIP](#) when dealing with a new MAC address, as this the address into the table. Aside from that, gt2 is used entirely the same as in infrastructure (regular IP) mode.

Sockets

A socket is an endpoint on the local machine that allows an application to communicate with other applications (through their own sockets) that are typically on remote machines, although they can also be on the local machine (the other application will often be referred to as the "remote machine", even though technically it may be the same machine). A single socket allows an application to both accept connections from remote machines and make connections to remote machines. For most applications, only one socket needs to be created. All incoming connections can be accepted on the socket, and all outgoing connections can be made using the socket.

A socket is created with the [gt2CreateSocket](#) function. If the function returns [GT2Success](#) then the socket was successfully created and bound to the local address (if one was provided). The socket that the "socket" parameter points to is valid until it is closed with [gt2CloseSocket](#), or an error is reported to the [gt2SocketErrorCallback](#) callback parameter. It is now ready to be used for making outgoing connections, and can be readied for allowing incoming connections by calling [gt2Listen](#) (see below). If the return result is anything other than [GT2Success](#), GT2 was unable to create the socket.

```
GT2Result gt2CreateSocket
(
    GT2Socket * socket,
    const char * localAddress,
    int outgoingBufferSize,
    int incomingBufferSize,
    gt2SocketErrorCallback callback
```

```
);
```

socket

This is a pointer to a [GT2Socket](#) variable where the socket will be stored.

localAddress

This is the address to bind to locally. Typically of the form ":<port>", e.g., ":7777". Can be NULL or "".

outgoingBufferSize

This is the byte size of the buffer for reliable outgoing messages.

This is a per-connection buffer. Can be 0 to use the internal default.

incomingBufferSize

This is the byte size of the buffer for out-of-order reliable incoming messages.

This is a per-connection buffer. Can be 0 to use the internal default.

callback

This callback is called if there is a fatal error with the socket.

gt2SocketErrorCallback

This callback is used to notify the application of a closed socket or fatal socket error condition. Once this callback returns, the socket and all of its connections are invalid and can no longer be used.

```
typedef void (* gt2SocketErrorCallback)
(
    GT2Socket socket
);
```

socket

The socket that had the error

gt2CloseSocket

This function is used to close a socket and any of its connections. Neither the socket nor any of its connections can be used once this call returns. The socket's will all be hard-closed (see [gt2CloseConnectionHard](#)).

```
void gt2CloseSocket  
(  
    GT2Socket socket  
);
```

socket

The socket to be closed.

gt2Think

Does any thinking for this socket and its connections. Callbacks are typically called from within this function (although they can also be called from other places). It is possible that during this think the socket or any of its connections may be closed, so care must be taken if calling other GT2 functions immediately after thinking. The more frequently this function is called, the faster GT2 will be able to respond (and reply to) messages. The general rule is to call it as frequently as you can, although calling it faster than every 10-20 milliseconds is probably unnecessary. If you are using [gt2Ping](#) to measure ping times, then the accuracy of the latency measurement will increase with the frequency at which this function is called.

```
void gt2Think  
(  
    GT2Socket socket  
);
```

socket

The socket to let think.

gt2Listen

If you want to be able to accept incoming connections from over the Internet, you must first create a socket, then start listening on it with `gt2Listen`. A `gt2ConnectAttemptCallback` is provided to handle possible incoming connection attempts. As soon as this function is called, the socket can start accepting incoming connections. If an attempt is made to connect to this socket after `gt2Listen` is called on it, the callback will be called. If this function is called with a `NULL` callback the socket stops listening for incoming connection attempts.

```
void gt2Listen
(
    GT2Socket socket,
    gt2ConnectionAttemptCallback callback
);
```

`socket`

The socket to start listening on.

`callback`

This callback is called when an incoming connection is attempted.

Can be `NULL` to refuse incoming connection attempts (the default).

gt2ConnectAttemptCallback

This notifies the socket that a remote system is attempting a connection. The IP and port of the remote system is provided, along with an optional initial message, and a latency estimate. These can be used to validate/authenticate the connecting system. This connection must either be accepted with `gt2Accept`, or rejected with `gt2Reject`. These can be called from within this callback, however they do not need to be. They can be called at any time after this callback is received. This is very useful for systems that need to check with another machine to authenticate the user (such as for a CDKey system). The latency is only an estimate, however it can be used for things such as only allowing low-ping or high-ping users onto a server.

```
typedef void (* gt2ConnectAttemptCallback)
```

```
(
    GT2Socket socket,
    GT2Connection connection,
    unsigned int ip,
    unsigned short port,
    int latency,
    GT2Byte * message,
    int len
);
```

socket

This is the socket to which someone is attempting to connect.

connection

This is the connection object for the incoming connection.

ip

The IP from which the connect attempt is coming.

port

The port from which the connect attempt is coming.

latency

An estimate of the round-trip time between the two machines (in milliseconds).

message

Optional initial data sent with the connect attempt. May be NULL.

len

Length of the initial data. May be 0.

gt2Accept

Accepts an incoming connection attempt. Once this has been called, the [GT2Connection](#) can be used normally. The connected callback member of the callbacks will be ignored, as it is only used when initiating a connection. If this returns [GT2False](#), that means the connection was closed between when the [gt2ConnectAttemptCallback](#) was called, and the connection was accepted. This would be caused by a remote close, or a time-out if it took too long to accept the connection. In this

case, the connection is closed and cannot be used.

```
GT2Bool gt2Accept
(
    GT2Connection connection,
    GT2ConnectionCallbacks * callbacks
);
```

connection

The connection being accepted

callbacks

The set of callbacks associated with the connection

See the [Connecting](#) section below for more information on [GT2ConnectionCallbacks](#).

gt2Reject

Use this call to reject an incoming connection. An optional rejection message can be sent. The connection is closed after this call and cannot be used.

```
void gt2Reject
(
    GT2Connection connection,
    const GT2Byte * message,
    int len
);
```

connection

The connection being rejected.

message

Rejection message. May be [NULL](#). Note that a 7 byte header needs to be accounted for.

len

Length of the rejection message. May be 0.

A len of -1 is equivalent to `(strlen(message) + 1)`

Connecting

The `gt2Connect` function is used to initiate a connection attempt to a remote socket on the Internet. After the remote socket is contacted, both it and the local connector will authenticate the other during a negotiation phase. Once the remote socket accepts the connection attempt, the connection will be established. The connection lasts until the closed callback gets called, which can happen because one side closed the connection with `gt2CloseConnection` (or `gt2CloseConnectionHard`), there was some sort of error on the connection, or the socket either connection uses is closed.

This call returns `GT2Success` if there are no problems starting the connection attempt, otherwise the return values signals the reason for the failure. If this call is blocking (blocking set to `GT2True`), then the return value signals the result of the entire connection attempt: `GT2Success` means the attempt succeeded, any other value means it failed. If the result is `GT2Success`, then the `GT2Connection` variable pointed to by the connection parameter will be set to this connection's `GT2Connection` object.

If this call is blocking, and it fails, the `GT2ConnectionCallbacks`'s connected callback may or may not be called. If there is some sort of initial failure (such as an error resolving the remote address, or allocating memory for the connection), the callback will not be called. If it fails after starting the negotiation process, then the callback will be called.

```
GT2Result gt2Connect
(
    GT2Socket socket,
    GT2Connection * connection,
    const char * remoteAddress,
    const GT2Byte * message,
    int len,
    int timeout,
```

```
GT2ConnectionCallbacks * callbacks,  
GT2Bool blocking  
);
```

socket

The socket to use to make the connection attempt.

connection

Pointer to the variable that the connection object will be stored in.

remoteAddress

The address to connect to. Must contain an IP/hostname and port.

Typically something like "myserver.someplace.com:12345"

message

Initial message. May be `NULL`. Note that a 7 byte header needs to be accounted for.

len

Length of the initial message. May be 0.

A len of -1 is equivalent to `(strlen(message) + 1)`

timeout

Time in milliseconds to wait before aborting the attempt.

If 0, keep trying until connected.

callbacks

The set of callbacks associated with the connection.

blocking

If `GPTrue`, don't return until the attempt has finished (success or failure).

```
typedef struct  
{  
    gt2ConnectedCallback connected;  
    gt2ReceivedCallback received;  
    gt2ClosedCallback closed;  
    gt2PingCallback ping;  
} GT2ConnectionCallbacks;
```

gt2ConnectedCallback

This callback is called when a connection attempt with [gt2Connect](#) finishes. If result is [GT2Success](#), then this connection attempt succeeded. The connection object can now be used for sending/receiving messages. Any other result indicates connection failure, and the connection object cannot be used again after this callback returns. If the result is [GT2Rejected](#), then message contains an optional rejection message sent by the listener. If result is not [GT2Rejected](#), then `message` will be `NULL` and `len` will be `0`.

```
typedef void (* gt2ConnectedCallback)
(
    GT2Connection connection,
    GT2ConnectResult result,
    GT2Byte * message,
    int len
);
```

`connection`

The connection that just finished connecting.

`result`

The result of the connect attempt. See *gt2.h* for all possible values.

Anything aside from [GT2Success](#) indicates failure.

`message`

If result is [GT2Rejected](#), this is the rejection message. May be `NULL`.

`len`

If result is [GT2Rejected](#), the length of the message. May be `0`.

gt2ReceivedCallback

This callback is called when a message is sent from the remote system with a [gt2Send](#). If the message is sent reliably, then it will always be received with this callback. If it is not sent reliably, then the message

might not arrive, or might arrive out of order.

```
typedef void (* gt2ReceivedCallback)
(
    GT2Connection connection,
    GT2Byte * message,
    int len,
    GT2Bool reliable
);
```

`connection`

The connection that received the message.

`message`

The message that was sent. May be NULL.

`len`

The length of the message. May be 0

`reliable`

Whether or not the message was sent reliably.

gt2ClosedCallback

This callback is called when the connection has been closed, which can be caused by either side calling `gt2CloseConnection` (or `gt2CloseConnectionHard`), either side closing the socket, or some sort of error. The connection cannot be used again once this callback returns.

```
typedef void (* gt2ClosedCallback)
(
    GT2Connection connection,
    GT2CloseReason reason
);
```

`connection`

The connection that was closed.

reason

The reason that the connection closed. See *gt2.h* for all possible values.

gt2PingCallback

This callback is called when a response to a ping sent on this connection is received. It gives a measure of the time it takes for a datagram to make a round-trip from one connection to the other. The latency reported in this callback will typically be larger than that reported by using ICMP pings between the two machines (the "ping" program uses ICMP pings), because ICMP pings happen at a lower level in the operating system. However, the ping reported in this callback will much more accurately reflect the latency of the application, as the application's messages must go through the same path as these pings, as opposed to ICMP.

Because pings are unreliable, a ping sent with [gt2Ping](#) is not guaranteed to make it through the entire round-trip. So not every call to [gt2Ping](#) will result in this callback being called. In addition, unreliable messages may be repeated (although this is a very rare occurrence), which means this callback could be called multiple times for a single call to [gt2Ping](#).

```
typedef void (* gt2PingCallback)
(
    GT2Connection connection,
    int latency
);
```

connection

The connection that the ping was sent and received on.

latency

The round-trip time for the ping, in milliseconds.

Sending

Once a connection has been established, messages can be sent back

and forth on it. To send a message, use the `gt2Send` function. If `message` is `NULL` or `len` is 0, then an empty message will be sent. When an empty message is received, `message` will be `NULL` and `len` will be 0. If the message is sent reliably, it is guaranteed to arrive, arrive only once, and arrive in order (relative to other reliable messages). If the message is sent unreliably, then it is not guaranteed to arrive, and if it does arrive, it is not guaranteed to arrive in order, or only once.

```
void gt2Send
(
    GT2Connection connection,
    const GT2Byte * message,
    int len,
    GT2Bool reliable
);
```

`connection`

The connection on which to send the message.

`message`

The message to send. May be `NULL`. Note that a 7 byte header needs to be accounted for if messages are reliable.

`len`

The length of the message. May be 0

`reliable`

Whether or not the message was sent reliably.

Closing Connections

There are two different ways a connection can be closed: they can be closed normally, or they can be "hard" closed. When a connection is closed normally, the connection's state is set to closing (i.e., `gt2GetConnectionState` will return `GT2Closing`), and a message is sent to the remote side telling it that the connection is closing. When confirmation is received that the remote side has received the message, the message is marked as closed (`gt2GetConnectionState` will return `GT2Closed`), the connection's closed callback is called, then the

connection is freed. Because this normal method of closing requires the closer to wait for confirmation from the remote side, the connection is not immediately fully closed or freed. If the connection is "hard" closed, then an (unreliable) message is sent to the remote side of the connection informing them that the connection is closed, the closed callback is called, then the connection is freed. Because it does not need to wait for confirmation, the connection can be freed sooner. However, if the message informing the remote side of the closure is lost, it may take the remote side some time to figure out that the connection was closed.

The remote side will typically find out either after trying to send a message that gets rejected locally (because the recipient has closed), or when the remote side's GT2 attempts to send a keep-alive message, which will also get rejected locally. The method to be used depends on the specifics of your application, but, in general, a normal close should be used when possible, as it will close the connection more gracefully, ensuring that both sides of the connection know that the connection is closed.

There are four functions that can be used for closing connections. Two of them do a normal close, and the other two do a hard close. Two of them close a single connection, and the other two close all of a socket's connections. The two functions that do hard closes will call the closed callback(s) from within the function, while the two that do normal closes will call the callback(s) at some later time.

```
void gt2CloseConnection(GT2Connection connection);  
void gt2CloseConnectionHard(GT2Connection connection);  
void gt2CloseAllConnections(GT2Socket socket);  
void gt2CloseAllConnectionsHard(GT2Socket socket);
```

`connection`

The connection to close.

`socket`

Close all of this socket's connections.

Filtering

GT2 allows an application to add one or more "filters" to any connection. These filters can either just monitor messages being sent and received, or they can actually modify the data before it gets sent or received. Any number of filters can be set on any connection, and the order of the filtering will be in the order they were added (oldest to newest). A filter is added by passing a callback to a function that adds the callback as either a send ([gt2AddSendFilter](#)) or receive ([gt2AddReceiveFilter](#)) filter. Then that callback will be called when a message is either sent or received (depending on what type of filter it is).

After a callback has been called, that filter it is responsible for letting GT2 know when its done with the message. This is done by calling either [gt2FilteredSend](#) for an outgoing message or [gt2FilteredReceive](#) for an incoming message. The filter has several options. If the filter does not call the appropriate function, then the message will be dropped (even if it was sent/received as reliable). The filter can call the appropriate function from within the callback with the same data that was passed into the callback. This will cause the message to continue without any modifications. Or, the filter can call the appropriate function with modified data, either from within the callback or at a later time.

gt2SendFilterCallback/gt2ReceiveFilterCallback

These are the filter callbacks, and are passed to [gt2AddSendFilter/gt2AddReceiveFilter](#) to be added as filters. The callbacks will be called in the order they were added. [gt2FilteredSend](#) or [gt2FilteredReceive](#) is typically called in response to one of these callbacks, either from within the callback, or at a later time. Note that if called after the callback has returned, the message pointer passed into the callback may no longer be valid. So if the message will be needed after the callback has returned, the data must be copied off.

```
typedef void (* gt2SendFilterCallback)
(
    GT2Connection connection,
    int filterID,
```

```

        const GT2Byte * message,
        int len,
        GT2Bool reliable
    );
    typedef void (* gt2ReceiveFilterCallback)
    (
        GT2Connection connection,
        int filterID,
        GT2Byte * message,
        int len,
        GT2Bool reliable
    );

```

connection

The connection on which the message is being sent or was received.

filterID

The filterID for this callback.

Must be passed to [gt2FilteredSend/gt2FilteredReceive](#).

Message

The message that was sent/received. May be NULL.

Note that for send, this is conts. but not receive.

len

The length of the message. May be 0.

reliable

Whether or not the message was sent or is being sent reliably.

gt2AddSendFilter/gt2AddReceiveFilter

These function are used to add a filter callback to the connection's filter list. The callback will get called with a message is either being sent or has been received. Callbacks will be called in the order they were added to the connection's filter list. These functions return [GT2False](#) if they were unable to add the filter to the list for any reason.

```
GT2Bool gt2AddSendFilter
(
    GT2Connection connection,
    gt2SendFilterCallback callback
);
GT2Bool gt2AddReceiveFilter
(
    GT2Connection connection,
    gt2ReceiveFilterCallback callback
);
```

connection

The connection on which the filter is being added.

callback

The callback to add to the filter list.

gt2RemoveSendFilter/gt2RemoveReceiveFilter

These functions are used to remove a filter callback from a connection's filter list. Filters should NOT be removed while a message is being filtered. If any are, filters could be skipped, or messages could be dropped. If the callback is NULL, all of the send or receive filters will be removed.

```
void gt2RemoveSendFilter
(
    GT2Connection connection,
    gt2SendFilterCallback callback
);
void gt2RemoveReceiveFilter
(
    GT2Connection connection,
    gt2ReceiveFilterCallback callback
);
```

connection

The connection on which the filter is being removed.

callback

The callback to remove from the filter list.

gt2FilteredSend/gt2FilteredReceive

These functions are used to pass on a message after a filter callback has been called. This will cause the message to either be passed to the next filter or, if this was the last filter, to be sent or received. If this is called from the filter callback, the message passed in can be the same message that was passed into the callback.

```
void gt2FilteredSend
(
    GT2Connection connection,
    int filterID,
    const GT2Byte * message,
    int len,
    GT2Bool reliable
);
void gt2FilteredReceive
(
    GT2Connection connection,
    int filterID,
    GT2Byte * message,
    int len,
    GT2Bool reliable
);
```

connection

The connection on which the message is being filtered.

filterID

This must be the same ID passed to the filter callback.

Message

The message being sent/received. May be **NULL**. Note that a 7 byte header needs to be accounted for when sending a reliable

message.

Note that for send, this is conts. but not receive.

len

The length of the message. May be 0.

reliable

For sending, this determines if the message should be sent reliably.

For receiving, this determines if the message was received reliably.

This value does not need to be the same value passed to the filter.

Encode/Decode

GT2 comes with an encode/decode sub-library that allows messages to be encoded with a format string into an array of bytes. For example, if a message consists of an int, a short, and a float, one function call can encode them into a 12 byte buffer (2 bytes for the message type, 4 for the int, 2 for the short, and 4 for the float). This array of bytes can then be sent as a regular GT2 message. On the other end of the connection, the message type can then be checked with `gtEncodedMessageType`. Once the correct type is determined, one function call can decode the 12 byte buffer into the original int, short, and float.

Format String

The format string used by the encoding function is simply a list of characters that signal what variable types are being encoded. For the full list of types, see *gt2Encode.h*. A sample format string for encoding an int, a short, a float, then a string would look like "iofs". GT2 supports encoding most of the standard C data types, regular C strings, wide strings, a "raw" array of bytes, and bits. If bits are adjacent in a format string, then they will be packed together.

gtEncodedMessageType

This function is used to determine the type of an encoded message stored in a buffer (such as a buffer passed to a

gt2ReceivedCallback.

```
GTMessageType gtEncodedMessageType  
(  
    char * inBuffer  
);
```

inBuffer

The buffer/message from which to get the type.

gtEncode[NoType[V]

These functions are used to encode the message. They take a format string, a buffer to encode into, a buffer size, and then all of the parameters to be encoded. For `gtEncode` and `gtEncodeNoType`, the parameters are passed on the end of the function, and for `gtEncodeV` and `gtEncodeNoTypeV`, the parameters are passed in as an args list. `gtEncode` and `gtEncodeV` take a message type to encode at the start of the buffer, while `gtEncodeNoType` and `gtEncodeNoTypeV` do not encode a type. This can be used for messages that have an unknown number of arguments. The first part of the message is encoded with a type, and it also contains information that lets the other end of the connection know what the rest of the message will look like. Then the rest of the message is encoded without a type. These functions return the number of bytes written to the buffer, or -1 if there is not enough space in the buffer to encode the entire message.

```
int gtEncode  
(  
    GTMessageType msgType,  
    const char * fmtString,  
    char * outBuffer,  
    int outLength,  
    ...  
);  
int gtEncodeV  
(
```

```

        GTMessageType msgType,
        const char * fmtString,
        char * outBuffer,
        int outLength,
        va_list * args
    );
int gtEncodeNoType
(
    const char * fmtString,
    char * outBuffer,
    int outLength,
    ...
);
int gtEncodeNoTypeV
(
    const char * fmtString,
    char * outBuffer,
    int outLength,
    va_list * args
);

```

msgType

The type to encode in the message.

fmtString

The format string that determines how the message is encoded.

outBuffer

The buffer to encode into.

outLength

The length of the outBuffer.

../args

The arguments that are encoded into the buffer according to the format string.

gtDecode[NoType][V]

These functions are used to decode an encoded message. They take a

format string, a buffer to decode from, a buffer size, and then a set of parameters to decode into (as with the scanf functions). For `gtDecode` and `gtDecodeNoType`, the parameters are passed on the end of the function, and for `gtDecodeV` and `gtDecodeNoTypeV`, the parameters are passed in as an args list. `gtDecode` and `gtDecodeV` will skip over a 2 bytes message type at the start of the buffer, while `gtDecodeNoType` and `gtDecodeNoTypeV` do not skip anything. This can be used for messages that have an unknown number of arguments. First the message type is checked with `gtEncodedMessageType`, and the first part of the message decoded with `gtDecode` or `gtDecodeV`. This part of the message can then be used to determine the format of the rest of the message, which can then be decoded with one or more calls to `gtDecodeNoType` or `gtDecodeNoTypeV`. These functions return the number of bytes read from the buffer, or -1 if there was a problem with the buffer.

```
int gtDecode
(
    const char * fmtString,
    char * inBuffer,
    int inLength,
    ...
);
int gtDecodeV
(
    const char * fmtString,
    char * inBuffer,
    int inLength,
    va_list * args
);
int gtDecodeNoType
(
    const char * fmtString,
    char * inBuffer,
    int inLength,
    ...
);
int gtDecodeNoTypeV
```

```
(
    const char * fmtString,
    char * inBuffer,
    int inLength,
    va_list * args
);
```

`fmtString`

The format string that determines how the message is decoded.

`inBuffer`

The buffer to decode from.

`inLength`

The length of the decode buffer.

`../args`

The decoded message parameters are stored in these arguments.

Socket Sharing

GT2 allows for a `GT2Socket` object to share its underlying socket, which allows it to be used for multiple purposes, such as using the socket for both GT2 and the GameSpy Query and Reporting SDK. The documentation below covers how the socket can be shared, see the Query and Reporting SDK documentation for the specifics on how to have it use the socket.

To get a `GT2Socket` object's underlying socket, use `gt2GetSocketSOCKET`. This socket will be valid until either `gt2CloseSocket` is called with the `GT2Socket`, or the `GT2Socket`'s `gt2SocketErrorCallback` gets called. For systems where `SOCKET` is not natively defined, it is defined in `nonport.h` (part of the GameSpy Common code), which is included by `gt2.h`.

```
SOCKET gt2GetSocketSOCKET
(
    GT2Socket socket
);
```

socket

The [GT2Socket](#) for which to get the underlying socket.

gt2SetUnrecognizedMessageCallback

This is used to set a callback to be called everytime a socket receives a message that it cannot match up to an existing connection. If a [GT2Socket](#) object's underlying socket is being shared, this allows an application to check for data that was not meant for GT2. See the documentation below for the callback for how to handle the data. If the callback parameter is [NULL](#), then any previously set callback will be removed.

```
void gt2SetUnrecognizedMessageCallback  
(  
    GT2Socket socket,  
    gt2UnrecognizedMessageCallback callback  
);
```

socket

This is the socket to which someone is attempting to connect.

callback

The callback to be called for unrecognized messages. May be [NULL](#).

gtUnrecognizedMessageCallback

This callback is called whenever a message is received that cannot be matched to an existing connection. The application must determine if the message was meant for it or not. If the application decides to handle the message, it should return [GT2True](#) from this function. This will tell the [GT2Socket](#) to ignore the message. If the application does not handle the message, it should return [GT2False](#). If it returns [GT2False](#), GT2 will send a message back to the machine that sent the original message, indicating that there is no existing connection for the message.

```
typedef GT2Bool (* gt2UnrecognizedMessageCallback)
(
    GT2Socket socket,
    unsigned int ip,
    unsigned short port,
    GT2Byte * message,
    int len
);
```

socket

This is the [GT2Socket](#) on which the message was received.

ip

The IP the message came from (in network byte order).

port

The port the remote machine (in host byte order).

message

The message contents. May be NULL.

len

The length of the message. May be 0.

Transport SDK Functions

gt2Accept	Accepts an incoming connection attempt.
gt2AddReceiveFilter	Adds a filter to the connection's incoming data filter list.
gt2AddressToString	Converts an IP and a port into a text string.
gt2AddSendFilter	Adds a filter to the connection's outgoing data filter list.
gt2CloseAllConnections	Closes all of a socket's connections.
gt2CloseAllConnectionsHard	Does a hard close on all of a socket's connections.
gt2CloseConnection	Starts closing a connection.
gt2CloseConnectionHard	Closes a connection immediately.
gt2CloseSocket	Closes a socket.
gt2Connect	Initiates a connection between a local socket and a remote socket.

gt2CreateAdHocSocket	Creates a new socket, which can be used for making outgoing connections or accepting incoming connections. See gt2CreateSocket for details.
gt2CreateSocket	Creates a new socket, which can be used for making outgoing connections or accepting incoming connections.
gt2FilteredReceive	Called in response to a gt2ReceiveFilterCallback being called. It can be called from within the callback, or at any later time.
gt2FilteredSend	Called in response to a gt2SendFilterCallback being called. It can be called from within the callback, or at any later time.
gt2GetConnectionData	Returns the user data pointer stored with this connection.
gt2GetConnectionSocket	Returns the socket which this connection exists on.
gt2GetConnectionState	Gets the connection's state.
gt2GetIncomingBufferFreeSpace	Gets the amount of available space in the connection's incoming buffer.

gt2GetIncomingBufferSize	Gets the total size of the connection's incoming buffer.
gt2GetLastSentMessageID	Gets the message id for the last reliably sent message. Unreliable messages do not have an id.
gt2GetLocalIP	Gets a socket's local IP.
gt2GetLocalPort	Get's a socket's local port.
gt2GetOutgoingBufferFreeSpace	Gets the amount of available space in the connection's outgoing buffer.
gt2GetOutgoingBufferSize	Gets the total size of the connection's outgoing buffer.
gt2GetRemoteIP	Gets the connection's remote IP.
gt2GetRemotePort	Get's the connection's remote port.
gt2GetSocketData	Returns the user data pointer stored with this socket.
gt2GetSocketSOCKET	This function returns the actual underlying socket for a GT2Socket.

gt2HostToNetworkInt	Convert an int from host to network byte order.
gt2HostToNetworkShort	Convert a short from host to network byte order.
gt2IPToAliases	Get the aliases associated with an IP address.
gt2IPToHostInfo	Looks up DNS host information based on an IP.
gt2IPToHostname	Get the hostname associated with an IP address.
gt2IPToIPs	Get the IPs associated with an IP address.
gt2Listen	Start (or stop) listening for incoming connections on a socket.
gt2NetworkToHostInt	Convert an int from network to host byte order.
gt2NetworkToHostShort	Convert a short from network to host byte order.
gt2Ping	Sends a ping on a connection in

	an attempt to determine latency.
gt2Reject	Rejects a connection attempt.
gt2RemoveReceiveFilter	Removes a filter from the connection's incoming data filter list.
gt2RemoveSendFilter	Removes a filter from the connection's outgoing data filter list.
gt2Send	Sends data over a connection, reliably or unreliably.
gt2SetConnectionData	Stores a user data pointer with this connection.
gt2SetReceiveDump	Sets a callback to which all incoming UDP packets are passed. This is at a lower level than the filters, can only be used for monitoring, and is designed for debugging purposes.
gt2SetSendDump	Sets a callback to which all outgoing UDP packets are passed. This is at a lower level than the filters, can only be used for monitoring, and is designed for debugging purposes.

gt2SetSocketData	Stores a user data pointer with this socket.
gt2SetUnrecognizedMessageCallback	Used to handle unrecognized messages, usually used for sharing a socket with another SDK.
gt2StringToAddress	Converts a string address, which is either a hostname ("www.gamespy.net") or a dotted IP ("1.2.3.4") into an IP and a port.
gt2StringToAliases	Get the aliases associated with a hostname or dotted IP.
gt2StringToHostInfo	Looks up DNS host information based on a hostname or dotted IP.
gt2StringToHostname	Get the hostname associated with a hostname or dotted IP.
gt2StringToIPs	Get the IPs associated with a hostname or dotted IP.
gt2Think	Does any thinking for this socket and its connections.
gt2WasMessageIDConfirmed	Checks if confirmation has been received that the remote end received a particular reliable

message.

[gti2IpToMac](#)

Converts a 32 bit IP address to a 48 bit Mac address

[gti2MacToIp](#)

Change mac ethernet to IP address.

gt2Accept

Accepts an incoming connection attempt.

```
GT2Bool gt2Accept(  
    GT2Connection connection,  
    GT2ConnectionCallbacks * callbacks );
```

Routine	Required Header	Distribution
gt2Accept	<gt2.h>	SDKZIP

Return Value

GT2False means the connection was closed between when the gt2ConnectAttemptCallback was called and this function was called. The connection cannot be used.

Parameters

connection

[in] The handle to the connection.

callbacks

[in] The set of callbacks associated with the connection.

Remarks

After a socket's `gt2ConnectAttemptCallback` has been called, this function can be used to accept the incoming connection attempt. It can be called from either within the callback or some later time. As soon as it is called the connection is active, and messages can be sent and received. The remote side of the connection will have its `connected` callback called with the result set to `GT2Success`. The callbacks that are passed in to this function are the same callbacks that get passed to `gt2Connect`, with the exception that the `connected` callback can be ignored, as the connection is already established. If this function returns `GT2True`, then the connection has been successfully accepted. If it returns `GT2False`, then the remote side has already closed the connection attempt. In that case, the connection is considered closed, and it cannot be referenced again.

Section Reference: [Gamespy Transport SDK](#)

See Also: [gt2Listen](#), [gt2ConnectAttemptCallback](#), [gt2Reject](#)

gt2AddReceiveFilter

Adds a filter to the connection's incoming data filter list.

```
GT2Bool gt2AddReceiveFilter(  
    GT2Connection connection,  
    gt2ReceiveFilterCallback callback );
```

Routine	Required Header	Distribution
gt2AddReceiveFilter	<gt2.h>	SDKZIP

Return Value

Returns `GT2False` if there was an error adding the filter (due to no free memory).

Parameters

connection

[in] The handle to the connection.

callback

[in] The filtering callback.

Remarks

The callback will get called when a message is being received. Callbacks will be called in the order they were added to the connection's filter list.

Section Reference: [Gamespy Transport SDK](#)

See Also: [gt2ReceiveFilterCallback](#), [gt2RemoveReceiveFilter](#), [gt2FilteredReceive](#)

gt2AddressToString

Converts an IP and a port into a text string.

```
const char * gt2AddressToString(  
    unsigned int ip,  
    unsigned short port,  
    char string[22] );
```

Routine	Required Header	Distribution
gt2AddressToString	<gt2.h>	SDKZIP

Return Value

The string is returned. If the string parameter is NULL, then an internal static string will be used. There are two internal strings that are alternated between.

Parameters

ip

[in] IP in network byte order. Can be 0.

port

[in] Port in host byte order. Can be 0.

string

[out] String will be placed in here. Can be NULL.

Remarks

The IP must be in network byte order, and the port in host byte order. The string must be able to hold at least 22 characters (including the NUL).

"123.123.123.123:12345"

If both the IP and port are non-zero, the string will be of the form

"1.2.3.4:5" ("`<IP>:<port>`").

If the port is zero, and the IP is non-zero, the string will be of the form

"1.2.3.4" ("`<IP>`").

If the IP is zero, and the port is non-zero, the string will be of the form "":5" ("`<port>`").

If both the IP and port are zero, the string will be an empty string ("").

Section Reference: [Gamespy Transport SDK](#)

See Also: [gt2StringToAddress](#)

gt2AddSendFilter

Adds a filter to the connection's outgoing data filter list.

```
GT2Bool gt2AddSendFilter(  
    GT2Connection connection,  
    gt2SendFilterCallback callback );
```

Routine	Required Header	Distribution
gt2AddSendFilter	<gt2.h>	SDKZIP

Return Value

Returns `GT2False` if there was an error adding the filter (due to no free memory).

Parameters

connection

[in] The handle to the connection.

callback

[in] The filtering callback.

Remarks

The callback will get called when a message is being sent. Callbacks will be called in the order they were added to the connection's filter list.

Section Reference: [Gamespy Transport SDK](#)

See Also: [gt2SendFilterCallback](#), [gt2RemoveSendFilter](#), [gt2FilteredSend](#)

gt2CloseAllConnections

Closes all of a socket's connections.

```
void gt2CloseAllConnections(  
    GT2Socket socket );
```

Routine	Required Header	Distribution
gt2CloseAllConnections	<gt2.h>	SDKZIP

Parameters

socket

[in] The handle to the socket.

Remarks

Same effect as calling `gt2CloseConnection` on all of the socket's connections.

Section Reference: [Gamespy Transport SDK](#)

See Also: [gt2CloseConnection](#), [gt2CloseConnectionHard](#), [gt2CloseAllConnectionsHard](#)

gt2CloseAllConnectionsHard

Does a hard close on all of a socket's connections.

```
void gt2CloseAllConnectionsHard(  
    GT2Socket socket );
```

Routine	Required Header	Distribution
gt2CloseAllConnectionsHard	<gt2.h>	SDKZIP

Parameters

socket

[in] The handle to the socket.

Remarks

Has the same effect as calling `gt2CloseConnectionHard` on all of the socket's connection.

Section Reference: [Gamespy Transport SDK](#)

See Also: [gt2CloseConnection](#), [gt2CloseConnectionHard](#), [gt2CloseAllConnections](#)

gt2CloseConnection

Starts closing a connection.

```
void gt2CloseConnection(  
    GT2Connection connection );
```

Routine	Required Header	Distribution
gt2CloseConnection	<gt2.h>	SDKZIP

Parameters

connection

[in] The handle to the connection.

Remarks

This function attempts to synchronize the close with the remote side of the connection. This means that the connection does not close immediately, and messages may be received while attempting the close. When the close is completed, the connection's closed callback will be called. Use **gt2CloseConnectionHard** to immediately close a connection.

Section Reference: [Gamespy Transport SDK](#)

See Also: [gt2CloseConnectionHard](#), [gt2CloseAllConnections](#), [gt2CloseAllConnectionsHard](#)

gt2CloseConnectionHard

Closes a connection immediately.

```
void gt2CloseConnectionHard(  
    GT2Connection connection );
```

Routine	Required Header	Distribution
gt2CloseConnectionHard	<gt2.h>	SDKZIP

Parameters

connection

[in] The handle to the connection.

Remarks

This function closes a connection without waiting for confirmation from the remote side of the connection. Messages in transit may be lost. The connection's closed callback will be called from within this function.

Section Reference: [Gamespy Transport SDK](#)

See Also: [gt2CloseConnection](#), [gt2CloseAllConnections](#), [gt2CloseAllConnectionsHard](#)

gt2CloseSocket

Closes a socket.

```
void gt2CloseSocket(  
    GT2Socket socket );
```

Routine	Required Header	Distribution
gt2CloseSocket	<gt2.h>	SDKZIP

Parameters

socket

[in] The handle to the socket.

Remarks

All existing connections will be hard closed, as if `gt2CloseAllConnectionsHard` was called for this socket. All connections send a close message to the remote side, and any closed callbacks will be called from within this function.

Section Reference: [Gamespy Transport SDK](#)

See Also: [gt2CreateSocket](#)

gt2Connect

Initiates a connection between a local socket and a remote socket.

```
GT2Result gt2Connect(  
    GT2Socket socket,  
    GT2Connection * connection,  
    const char * remoteAddress,  
    const GT2Byte * message,  
    int len,  
    int timeout,  
    GT2ConnectionCallbacks * callbacks,  
    GT2Bool blocking );
```

Routine	Required Header	Distribution
gt2Connect	<gt2.h>	SDKZIP

Return Value

If blocking is true, GT2Success means the connect attempt succeeded, and anything else means it failed.

If blocking is false, GT2Success means the connection is being attempted, and anything else means there was an error and the attempt has been aborted.

Parameters

socket

[in] The handle to the socket.

connection

[out] A pointer to where the connection handle will be stored.

remoteAddress

[in] The address to connect to.

message

[in] An optional initial message (may be NULL).

len

[in] Length of the initial message (may be 0, or -1 for strlen)

timeout

[in] Timeout in milliseconds (may be 0 for infinite retries)

callbacks

[in] GT2Connection related callbacks.

blocking

[in] If GT2True, don't return until the attempt has completed (successfully or unsuccessfully).

Remarks

The **gt2Connect** function is used to initiate a connection attempt to a remote socket on the Internet. After the remote socket is contacted, both it and the local connector will authenticate the other during a negotiation phase. Once the remote socket accepts the connection attempt, the connection will be established. The connection lasts until the closed callback gets called, which can happen because one side closed the connection with `gt2CloseConnection` (or `gt2CloseConnectionHard`), there was some sort of error on the connection, or the socket either connection uses is closed.

This call returns `GT2Success` if there are no problems starting the connection attempt, otherwise the return values signals the reason for the failure. If this call is blocking (blocking set to `GT2True`), then the return value signals the result of the entire connection attempt: `GT2Success` means the attempt succeeded, any other value means it failed. If the result is `GT2Success`, then the **gt2Connection** variable pointed to by the connection parameter will be set to this connection's **gt2Connection** object. If this call is blocking, and it fails, the **gt2ConnectionCallbacks**'s connected callback may or may not be called. If there is some sort of initial failure (such as an error resolving the remote address, or allocating memory for the connection), the callback will not be called. If it fails after starting the negotiation process, then the callback will be called. Note that the 7 byte header must be accounted for in the message since the connection message is sent reliably.

Section Reference: [Gamespy Transport SDK](#)

See Also: [gt2ConnectedCallback](#), [gt2ClosedCallback](#), [gt2CloseConnection](#), [gt2AddressToString](#)

gt2CreateSocket

Creates a new socket, which can be used for making outgoing connections or accepting incoming connections.

```
GT2Result gt2CreateSocket(  
    GT2Socket * socket,  
    const char * localAddress,  
    int outgoingBufferSize,  
    int incomingBufferSize,  
    gt2SocketErrorCallback callback );
```

Routine	Required Header	Distribution
gt2CreateSocket	<gt2.h>	SDKZIP

Return Value

If the function returns GT2Success then the socket was successfully created. Otherwise, GT2 was unable to create the socket.

Parameters

socket

[out] Pointer to the socket handle.

localAddress

[in] The address to bind to locally. Typically of the form ":<port>", e.g., ":7777". Can be NULL or "".

outgoingBufferSize

[in] The byte size of the per-connection buffer for reliable outgoing messages. Can be 0 to use the internal default.

incomingBufferSize

[in] The byte size of the per-connection buffer for out-of-order reliable incoming messages. Can be 0 to use the internal default.

callback

[in] The callback to be called if there is a fatal error with the socket.

Remarks

A socket is an endpoint on the local machine that allows an application to communicate with other applications (through their own sockets) that are typically on remote machines, although they can also be on the local machine (the other application will often be referred to as the "remote machine", even though technically it may be the same machine). A single socket allows an application to both accept connections from remote machines and make connections to remote machines. For most applications, only one socket needs to be created. All incoming connections can be accepted on the socket, and all outgoing connections can be made using the socket. A socket is created with the **gt2CreateSocket** function. If the function returns GT2Success then the socket was successfully created and bound to the local address (if one was provided). The socket that the "socket" parameter points to is valid until it is closed with gt2CloseSocket, or an error is reported to the gt2SocketErrorCallback callback parameter. It is now ready to be used for making outgoing connections, and can be readied for allowing incoming connections by calling gt2Listen. If the return result is anything other than GT2Success, GT2 was unable to create the socket.

Section Reference: [Gamespy Transport SDK](#)

See Also: [gt2SocketErrorCallback](#), [gt2CloseSocket](#), [gt2Listen](#), [gt2Connect](#)

gt2FilteredReceive

Called in response to a gt2ReceiveFilterCallback being called. It can be called from within the callback, or at any later time.

```
void gt2FilteredReceive(  
    GT2Connection connection,  
    int filterID,  
    GT2Byte * message,  
    int len,  
    GT2Bool reliable );
```

Routine	Required Header	Distribution
gt2FilteredReceive	<gt2.h>	SDKZIP

Parameters

connection

[in] The handle to the connection.

filterID

[in] The ID passed to the `gt2ReceiveFilterCallback`.

message

[in] The message that was received. May be NULL.

len

[in] The length of the message in bytes. May be 0.

reliable

[in] True if this is a reliable message.

Remarks

Used to pass on a message after a filter callback has been called. This will cause the message to either be passed to the next filter or, if this was the last filter, to be received. If this is called from the filter callback, the message passed in can be the same message that was passed into the callback.

Section Reference: [Gamespy Transport SDK](#)

See Also: [gt2ReceiveFilterCallback](#), [gt2AddReceiveFilter](#), [gt2RemoveReceiveFilter](#)

gt2FilteredSend

Called in response to a gt2SendFilterCallback being called. It can be called from within the callback, or at any later time.

```
void gt2FilteredSend(  
    GT2Connection connection,  
    int filterID,  
    GT2Byte * message,  
    int len,  
    GT2Bool reliable );
```

Routine	Required Header	Distribution
gt2FilteredSend	<gt2.h>	SDKZIP

Parameters

connection

[in] The handle to the connection.

filterID

[in] The ID passed to the `gt2SendFilterCallback`.

message

[in] The message that was sent. May be NULL.

len

[in] The length of the message in bytes. May be 0.

reliable

[in] True if this is a reliable message.

Remarks

Used to pass on a message after a filter callback has been called. This will cause the message to either be passed to the next filter or, if this was the last filter, to be sent. If this is called from the filter callback, the message passed in can be the same message that was passed into the callback. Note that the 7 byte header must be accounted for in the message if the function sends the message reliably.

Section Reference: [Gamespy Transport SDK](#)

See Also: [gt2SendFilterCallback](#), [gt2AddSendFilter](#), [gt2RemoveSendFilter](#)

gt2GetConnectionData

Returns the user data pointer stored with this connection.

```
void * gt2GetConnectionData(  
    GT2Connection connection );
```

Routine	Required Header	Distribution
gt2GetConnectionData	<gt2.h>	SDKZIP

Return Value

A pointer to this connection's user data.

Parameters

connection

[in] The handle to the connection.

Remarks

Each connection has a user data pointer associated with it that can be used by the application for any purpose.

Section Reference: [Gamespy Transport SDK](#)

See Also: [gt2SetSocketData](#), [gt2GetSocketData](#), [gt2SetConnectionData](#)

gt2GetConnectionSocket

Returns the socket which this connection exists on.

```
GT2Socket gt2GetConnectionSocket(  
    GT2Connection connection );
```

Routine	Required Header	Distribution
gt2GetConnectionSocket	<gt2.h>	SDKZIP

Return Value

The socket on which the connection was created or accepted.

Parameters

connection

[in] The handle to the connection.

Remarks

All connections are created through either `gt2Connect` or `gt2ConnectAttemptCallback`. This function will return the socket associated with the connection.

Section Reference: [Gamespy Transport SDK](#)

See Also: [gt2Connect](#), [gt2ConnectAttemptCallback](#)

gt2GetConnectionState

Gets the connection's state.

```
GT2ConnectionState gt2GetConnectionState(  
    GT2Connection connection );
```

Routine	Required Header	Distribution
gt2GetConnectionState	<gt2.h>	SDKZIP

Return Value

GT2Connecting, GT2Connected, GT2Closing, or GT2Closed

Parameters

connection

[in] The handle to the connection.

Remarks

A connection is either connecting, connected, closing, or closed.

GT2Connecting - the connection is still being negotiated

GT2Connected - the connection is active (has successfully connected, and not yet closing)

GT2Closing - the connection is in the process of closing (sent a close message and waiting for confirmation)

GT2Closed - the connection has already been closed and will soon be freed.

Section Reference: [Gamespy Transport SDK](#)

gt2GetIncomingBufferFreeSpace

Gets the amount of available space in the connection's incoming buffer.

```
int gt2GetIncomingBufferFreeSpace(  
    GT2Connection connection );
```

Routine	Required Header	Distribution
gt2GetIncomingBufferFreeSpace	<gt2.h>	SDKZIP

Return Value

The size in bytes of the free space in the connection's incoming buffer.

Parameters

connection

[in] The handle to the connection.

Section Reference: [Gamespy Transport SDK](#)

See Also: [gt2CreateSocket](#), [gt2GetIncomingBufferSize](#),
[gt2GetOutgoingBufferSize](#), [gt2GetOutgoingBufferFreeSpace](#)

gt2GetIncomingBufferSize

Gets the total size of the connection's incoming buffer.

```
int gt2GetIncomingBufferSize(  
    GT2Connection connection );
```

Routine	Required Header	Distribution
gt2GetIncomingBufferSize	<gt2.h>	SDKZIP

Return Value

The size in bytes of the connection's incoming buffer.

Parameters

connection

[in] The handle to the connection.

Section Reference: [Gamespy Transport SDK](#)

See Also: [gt2CreateSocket](#), [gt2GetIncomingBufferFreeSpace](#), [gt2GetOutgoingBufferSize](#), [gt2GetOutgoingBufferFreeSpace](#)

gt2GetLocalIP

Gets a socket's local IP.

```
unsigned int gt2GetLocalIP(  
    GT2Socket socket );
```

Routine	Required Header	Distribution
gt2GetLocalIP	<gt2.h>	SDKZIP

Return Value

The local IP in network byte order.

Parameters

socket

[in] The handle to the socket.

Section Reference: [Gamespy Transport SDK](#)

See Also: [gt2GetRemoteIP](#), [gt2GetRemotePort](#), [gt2GetLocalPort](#)

gt2GetLocalPort

Get's a socket's local port.

```
unsigned short gt2GetLocalPort(  
    GT2Socket socket );
```

Routine	Required Header	Distribution
gt2GetLocalPort	<gt2.h>	SDKZIP

Return Value

The local port in host byte order.

Parameters

socket

[in] The handle to the socket.

Section Reference: [Gamespy Transport SDK](#)

See Also: [gt2GetRemoteIP](#), [gt2GetRemotePort](#), [gt2GetLocalIP](#)

gt2GetOutgoingBufferFreeSpace

Gets the amount of available space in the connection's outgoing buffer.

```
int gt2GetOutgoingBufferFreeSpace(  
    GT2Connection connection );
```

Routine	Required Header	Distribution
gt2GetOutgoingBufferFreeSpace	<gt2.h>	SDKZIP

Return Value

The size in bytes of the free space in the connection's outgoing buffer.

Parameters

connection

[in] The handle to the connection.

Section Reference: [Gamespy Transport SDK](#)

See Also: [gt2CreateSocket](#), [gt2GetIncomingBufferSize](#),
[gt2GetIncomingBufferFreeSpace](#), [gt2GetOutgoingBufferSize](#)

gt2GetOutgoingBufferSize

Gets the total size of the connection's outgoing buffer.

```
int gt2GetOutgoingBufferSize(  
    GT2Connection connection );
```

Routine	Required Header	Distribution
gt2GetOutgoingBufferSize	<gt2.h>	SDKZIP

Return Value

The size in bytes of the connection's outgoing buffer.

Parameters

connection

[in] The handle to the connection.

Section Reference: [Gamespy Transport SDK](#)

See Also: [gt2CreateSocket](#), [gt2GetIncomingBufferSize](#),
[gt2GetIncomingBufferFreeSpace](#), [gt2GetOutgoingBufferFreeSpace](#)

gt2GetRemoteIP

Gets the connection's remote IP.

```
unsigned int gt2GetRemoteIP(  
    GT2Connection connection );
```

Routine	Required Header	Distribution
gt2GetRemoteIP	<gt2.h>	SDKZIP

Return Value

The remote IP in network byte order.

Parameters

connection

[in] The handle to the connection.

Remarks

Gets the IP of the computer on the remote side of the connection.

Section Reference: [Gamespy Transport SDK](#)

See Also: [gt2GetRemotePort](#), [gt2GetLocalIP](#), [gt2GetLocalPort](#)

gt2GetRemotePort

Get's the connection's remote port.

```
unsigned short gt2GetRemotePort(  
    GT2Connection connection );
```

Routine	Required Header	Distribution
gt2GetRemotePort	<gt2.h>	SDKZIP

Return Value

The remote port in host byte order.

Parameters

connection

[in] The handle to the connection.

Remarks

Gets the port of the computer on the remote side of the connection.

Section Reference: [Gamespy Transport SDK](#)

See Also: [gt2GetRemoteIP](#), [gt2GetLocalIP](#), [gt2GetLocalPort](#)

gt2GetSocketData

Returns the user data pointer stored with this socket.

```
void * gt2GetSocketData(  
    GT2Socket socket );
```

Routine	Required Header	Distribution
gt2GetSocketData	<gt2.h>	SDKZIP

Return Value

A pointer to this socket's user data.

Parameters

socket

[in] The handle to the socket.

Remarks

Each socket has a user data pointer associated with it that can be used by the application for any purpose.

Section Reference: [Gamespy Transport SDK](#)

See Also: [gt2SetSocketData](#), [gt2SetConnectionData](#), [gt2GetConnectionData](#)

gt2GetSocketSOCKET

This function returns the actual underlying socket for a GT2Socket.

```
SOCKET gt2GetSocketSOCKET(  
    GT2Socket socket );
```

Routine	Required Header	Distribution
gt2GetSocketSOCKET	<gt2.h>	SDKZIP

Return Value

The underlying socket associated with the GT2Socket.

Parameters

socket

[in] The handle to the socket.

Remarks

This can be used for socket sharing purposes, along with the `gt2UnrecognizedMessageCallback`.

Section Reference: [Gamespy Transport SDK](#)

See Also: [gt2SetUnrecognizedMessageCallback](#)

gt2HostToNetworkInt

Convert an int from host to network byte order.

```
unsigned int gt2HostToNetworkInt(  
    unsigned int i );
```

Routine	Required Header	Distribution
gt2HostToNetworkInt	<gt2.h>	SDKZIP

Return Value

The int in network byte order.

Parameters

i

[in] Int to convert.

Remarks

This is a utility function to help deal with byte order differences for multi-platform applications. Convert from host to network byte order before sending over the network, then convert back to host byte order when receiving.

Section Reference: [Gamespy Transport SDK](#)

See Also: [gt2NetworkToHostInt](#), [gt2NetworkToHostShort](#), [gt2HostToNetworkShort](#)

gt2HostToNetworkShort

Convert a short from host to network byte order.

```
unsigned short gt2HostToNetworkShort(  
    unsigned short s );
```

Routine	Required Header	Distribution
gt2HostToNetworkShort	<gt2.h>	SDKZIP

Return Value

The short in network byte order.

Parameters

s

[in] Short to convert.

Remarks

This is a utility function to help deal with byte order differences for multi-platform applications. Convert from host to network byte order before sending over the network, then convert back to host byte order when receiving.

Section Reference: [Gamespy Transport SDK](#)

See Also: [gt2NetworkToHostInt](#), [gt2HostToNetworkInt](#), [gt2NetworkToHostShort](#)

gt2IPToAliases

Get the aliases associated with an IP address.

```
char ** gt2IPToAliases(  
    unsigned int ip );
```

Routine	Required Header	Distribution
gt2IPToAliases	<gt2.h>	SDKZIP

Return Value

Aliases associated with the IP address.

Parameters

ip

[in] IP to lookup, in network byte order.

Remarks

This is a utility function which provides a subset of the functionality of `gt2IPToHostInfo`. See the `gt2IPToHostInfo` documentation for important details.

Section Reference: [Gamespy Transport SDK](#)

See Also: [gt2IPToHostInfo](#)

gt2IPToHostInfo

Looks up DNS host information based on an IP.

```
const char * gt2IPToHostInfo(  
    unsigned int ip,  
    char *** aliases,  
    unsigned int *** ips );
```

Routine	Required Header	Distribution
gt2IPToHostInfo	<gt2.h>	SDKZIP

Return Value

The hostname associated with the IP, or NULL if no information was available for the host.

Parameters

ip

[in] IP to look up, in network byte order.

aliases

[out] On success, the variable passed in will point to a NULL-terminated list of alternate names for the host. Can be NULL.

ips

[out] On success, the variable passed in will point to a NULL-terminated list of alternate IPs for the host. Can be NULL.

Remarks

If the function can successfully lookup the host's info, the host's main hostname will be returned. If it cannot find the host's info, it returns NULL. For the aliases parameter, pass in a pointer to a variable of type (char **). If this parameter is not NULL, and the function succeeds, the variable will point to a NULL-terminated list of alternate names for the host.

For the ips parameter, pass in a pointer to a variable of type (int **). If this parameter is not NULL, and the function succeeds, the variable will point to a NULL-terminated list of alternate IPs for the host. Each element in the list is actually a pointer to an unsigned int, which is an IP address in network byte order.

The return value, aliases, and IPs all point to an internal data structure, and none of these values should be modified directly. Also, the data is only valid until another call needs to use the same data structure (virtually every internet address function will use this data structure). If the data will be needed in the future, it should be copied off.

This function may need to contact a DNS server, which can cause the function to block for an indefinite period of time. Usually it is < 2 seconds, but on certain systems, and under certain circumstances, it can take 30 seconds or longer.

Section Reference: [Gamespy Transport SDK](#)

See Also: [gt2StringToHostInfo](#), [gt2IPToHostname](#), [gt2IPToAliases](#), [gt2IPToIPs](#)

gt2IPToHostname

Get the hostname associated with an IP address.

```
const char * gt2IPToHostname(  
    unsigned int ip );
```

Routine	Required Header	Distribution
gt2IPToHostname	<gt2.h>	SDKZIP

Return Value

Hostname associated with the IP address.

Parameters

ip

[in] IP to lookup, in network byte order.

Remarks

This is a utility function which provides a subset of the functionality of `gt2IPToHostInfo`. See the `gt2IPToHostInfo` documentation for important details.

Section Reference: [Gamespy Transport SDK](#)

See Also: [gt2IPToHostInfo](#)

gt2IPToIPs

Get the IPs associated with an IP address.

```
unsigned int ** gt2IPToIPs(  
    unsigned int ip );
```

Routine	Required Header	Distribution
gt2IPToIPs	<gt2.h>	SDKZIP

Return Value

IPs associated with the IP address.

Parameters

ip

[in] IP to lookup, in network byte order.

Remarks

This is a utility function which provides a subset of the functionality of `gt2IPToHostInfo`. See the `gt2IPToHostInfo` documentation for important details.

Section Reference: [Gamespy Transport SDK](#)

See Also: [gt2IPToHostInfo](#)

gt2Listen

Start (or stop) listening for incoming connections on a socket.

```
void gt2Listen(  
    GT2Socket socket,  
    gt2ConnectAttemptCallback callback );
```

Routine	Required Header	Distribution
gt2Listen	<gt2.h>	SDKZIP

Parameters

socket

[in] The handle to the socket.

callback

[in] Function to be called when the operation completes

Remarks

Once a socket starts listening, any connections attempts will cause the callback to be called.

If the socket is already listening, this callback will replace the existing callback being used.

If the callback is NULL, this will cause the connection to stop listening.

Section Reference: [Gamespy Transport SDK](#)

See Also: [gt2CreateSocket](#), [gt2ConnectAttemptCallback](#)

gt2NetworkToHostInt

Convert an int from network to host byte order.

```
unsigned int gt2NetworkToHostInt(  
    unsigned int i );
```

Routine	Required Header	Distribution
gt2NetworkToHostInt	<gt2.h>	SDKZIP

Return Value

The int in host byte order.

Parameters

i

[in] Int to convert.

Remarks

This is a utility function to help deal with byte order differences for multi-platform applications. Convert from host to network byte order before sending over the network, then convert back to host byte order when receiving.

Section Reference: [Gamespy Transport SDK](#)

See Also: [gt2HostToNetworkInt](#), [gt2NetworkToHostShort](#), [gt2HostToNetworkShort](#)

gt2NetworkToHostShort

Convert a short from network to host byte order.

```
unsigned short gt2NetworkToHostShort(  
    unsigned short s );
```

Routine	Required Header	Distribution
gt2NetworkToHostShort	<gt2.h>	SDKZIP

Return Value

The short in host byte order.

Parameters

s

[in] Short to convert.

Remarks

This is a utility function to help deal with byte order differences for multi-platform applications. Convert from host to network byte order before sending over the network, then convert back to host byte order when receiving.

Section Reference: [Gamespy Transport SDK](#)

See Also: [gt2NetworkToHostInt](#), [gt2HostToNetworkInt](#), [gt2HostToNetworkShort](#)

gt2Ping

Sends a ping on a connection in an attempt to determine latency.

```
void gt2Ping(  
    GT2Connection connection );
```

Routine	Required Header	Distribution
gt2Ping	<gt2.h>	SDKZIP

Parameters

connection

[in] The handle to the connection.

Remarks

The ping callback will, which is set as part of the GT2ConnectionCallbacks in either gt2Connect or gt2Accept, will be called if and when a ping finishes making a round-trip between the local end of the connection and the remote end. The ping is unreliable, and either it or the pong sent in reply could be dropped, resulting in the callback never being called. Or it could even arrive multiple times, resulting in multiple calls to the callback (this case is very rare).

Section Reference: [Gamespy Transport SDK](#)

See Also: [gt2PingCallback](#)

gt2Reject

Rejects a connection attempt.

```
void gt2Reject(  
    GT2Connection connection,  
    const GT2Byte * message,  
    int len );
```

Routine	Required Header	Distribution
gt2Reject	<gt2.h>	SDKZIP

Parameters

connection

[in] The handle to the connection.

message

[in] Rejection message. May be NULL.

len

[in] Length of the rejection message. May be 0. A len of -1 is equivalent to $(\text{strlen}(\text{message}) + 1)$.

Remarks

After a socket's `gt2ConnectAttemptCallback` has been called, this function can be used to reject the incoming connection attempt. It can be called from either within the callback or some later time. Once the function is called the connection is considered closed and cannot be referenced again. The remote side attempting the connection will have its connected callback called with the result set to **gt2Rejected**. If the message is not NULL and the len is not 0, the message will be sent with the rejection, and passed into the remote side's connected callback. Note that the 7 byte header must be accounted for in the message since this function will send the rejection message reliably.

Section Reference: [Gamespy Transport SDK](#)

See Also: [gt2Listen](#), [gt2ConnectAttemptCallback](#), [gt2Accept](#)

gt2RemoveReceiveFilter

Removes a filter from the connection's incoming data filter list.

```
void gt2RemoveReceiveFilter(  
    GT2Connection connection,  
    gt2ReceiveFilterCallback callback );
```

Routine	Required Header	Distribution
gt2RemoveReceiveFilter	<gt2.h>	SDKZIP

Parameters

connection

[in] The handle to the connection.

callback

[in] The filtering callback to remove. NULL removes all filters.

Remarks

Filters should not be removed while a message is being filtered. If the callback is NULL, all of the filters will be removed.

Section Reference: [Gamespy Transport SDK](#)

See Also: [gt2ReceiveFilterCallback](#), [gt2AddReceiveFilter](#), [gt2FilteredReceive](#)

gt2RemoveSendFilter

Removes a filter from the connection's outgoing data filter list.

```
void gt2RemoveSendFilter(  
    GT2Connection connection,  
    gt2SendFilterCallback callback );
```

Routine	Required Header	Distribution
gt2RemoveSendFilter	<gt2.h>	SDKZIP

Parameters

connection

[in] The handle to the connection.

callback

[in] The filtering callback to remove. NULL removes all filters.

Remarks

Filters should not be removed while a message is being filtered. If the callback is NULL, all of the filters will be removed.

Section Reference: [Gamespy Transport SDK](#)

See Also: [gt2SendFilterCallback](#), [gt2AddSendFilter](#), [gt2FilteredSend](#)

gt2Send

Sends data over a connection, reliably or unreliably.

```
void gt2Send(  
    GT2Connection connection,  
    const GT2Byte * message,  
    int len,  
    GT2Bool reliable );
```

Routine	Required Header	Distribution
gt2Send	<gt2.h>	SDKZIP

Parameters

connection

[in] The handle to the connection.

message

[in] The message to send. Can be NULL.

len

[in] The length of the message. Can be 0. A len of -1 is equivalent to $(\text{strlen}(\text{message}) + 1)$.

reliable

[in] if GT2True, send the message reliably, otherwise send it unreliably.

Remarks

Once a connection has been established, messages can be sent back and forth on it. To send a message, use the **gt2Send** function. If message is NULL or len is 0, then an empty message will be sent. When an empty message is received, message will be NULL and len will be 0. If the message is sent reliably, it is guaranteed to arrive, arrive only once, and arrive in order (relative to other reliable messages). If the message is sent unreliably, then it is not guaranteed to arrive, and if it does arrive, it is not guaranteed to arrive in order, or only once. Note that the 7 byte header must be accounted for in the message if the function sends the message reliably.

Section Reference: [Gamespy Transport SDK](#)

See Also: [gt2ReceivedCallback](#)

gt2SetConnectionData

Stores a user data pointer with this connection.

```
void gt2SetConnectionData(  
    GT2Connection connection,  
    void * data );
```

Routine	Required Header	Distribution
gt2SetConnectionData	<gt2.h>	SDKZIP

Parameters

connection

[in] The handle to the connection.

data

[in] A pointer to this connection's user data.

Remarks

Each connection has a user data pointer associated with it that can be used by the application for any purpose.

Section Reference: [Gamespy Transport SDK](#)

See Also: [gt2SetSocketData](#), [gt2GetSocketData](#), [gt2GetConnectionData](#)

gt2SetReceiveDump

Sets a callback to which all incoming UDP packets are passed. This is at a lower level than the filters, can only be used for monitoring, and is designed for debugging purposes.

```
void gt2SetReceiveDump(  
    GT2Socket socket,  
    gt2DumpCallback callback );
```

Routine	Required Header	Distribution
gt2SetReceiveDump	<gt2.h>	SDKZIP

Parameters

socket

[in] The handle to the socket.

callback

[in] The dump callback to set.

Remarks

Sets a callback to be called whenever a UDP datagram or a connection reset is received. Pass in a callback of NULL to remove the callback. The dump sits at a lower level than the filters, and allow an app to keep an eye on exactly what datagrams are being received, allowing for close monitoring. The dump cannot be used to modify data, only monitor it. The dump is useful for debugging purposes, and to keep track of data receive rates (e.g., the Quake 3 engine's netgraph). Note that these are the actual UDP datagrams being received - datagrams may be dropped, repeated, or out-of-order. Control datagrams (those used internally by the protocol) will be passed to the dump callback, and certain application messages will have a header at the beginning.

Section Reference: [Gamespy Transport SDK](#)

See Also: [gt2DumpCallback](#), [gt2SetSendDump](#)

gt2SetSendDump

Sets a callback to which all outgoing UDP packets are passed. This is at a lower level than the filters, can only be used for monitoring, and is designed for debugging purposes.

```
void gt2SetSendDump(  
    GT2Socket socket,  
    gt2DumpCallback callback );
```

Routine	Required Header	Distribution
gt2SetSendDump	<gt2.h>	SDKZIP

Parameters

socket

[in] The handle to the socket.

callback

[in] The dump callback to set.

Remarks

Sets a callback to be called whenever a UDP datagram is sent. Pass in a callback of NULL to remove the callback. The dump sits at a lower level than the filters, and allows an app to keep an eye on exactly what datagrams are being sent, allowing for close monitoring. The dump cannot be used to modify data, only monitor it. The dump is useful for debugging purposes, and to keep track of data send rates (e.g., the Quake 3 engine's netgraph). Note that these are the actual UDP datagrams being sent - datagrams may be dropped, repeated, or out-of-order. Control datagrams (those used internally by the protocol) will be passed to the dump callback, and certain application messages will have a header at the beginning.

Section Reference: [Gamespy Transport SDK](#)

See Also: [gt2DumpCallback](#), [gt2SetReceiveDump](#)

gt2SetSocketData

Stores a user data pointer with this socket.

```
void gt2SetSocketData(  
    GT2Socket socket,  
    void * data );
```

Routine	Required Header	Distribution
gt2SetSocketData	<gt2.h>	SDKZIP

Parameters

socket

[in] The handle to the socket.

data

[in] A pointer to this socket's user data.

Remarks

Each socket has a user data pointer associated with it that can be used by the application for any purpose.

Section Reference: [Gamespy Transport SDK](#)

See Also: [gt2GetSocketData](#), [gt2SetConnectionData](#), [gt2GetConnectionData](#)

gt2SetUnrecognizedMessageCallback

Used to handle unrecognized messages, usually used for sharing a socket with another SDK.

```
void gt2SetUnrecognizedMessageCallback(  
    GT2Socket socket,  
    gt2UnrecognizedMessageCallback callback );
```

Routine	Required Header	Distribution
gt2SetUnrecognizedMessageCallback	<gt2.h>	SDKZIP

Parameters

socket

[in] The handle to the socket.

callback

[in] Function to be called when an unrecognized message is received.

Remarks

This is used to set a callback to be called everytime a socket receives a message that it cannot match up to an existing connection. If a GT2Socket object's underlying socket is being shared, this allows an application to check for data that was not meant for GT2. If the callback parameter is NULL, then any previously set callback will be removed. This is typically used when you are sharing a GT2Socket with another SDK, such as QR2 or NAT Negotiation. Setting an unrecognized callback allows you to pass messages meant for another SDK to the appropriate place.

Section Reference: [Gamespy Transport SDK](#)

See Also: [gt2UnrecognizedMessageCallback](#), [gt2GetSocketSOCKET](#)

gt2StringToAddress

Converts a string address, which is either a hostname ("www.gamespy.net") or a dotted IP ("1.2.3.4") into an IP and a port.

```
GT2Bool gt2StringToAddress(  
    const char * string,  
    unsigned int * ip,  
    unsigned short * port );
```

Routine	Required Header	Distribution
gt2StringToAddress	<gt2.h>	SDKZIP

Return Value

Returns `GT2False` if there was an error parsing the string, or if a supplied hostname can't be resolved.

Parameters

string

[in] String to convert.

ip

[out] IP in network byte order. Can be NULL.

port

[out] Port in host byte order. Can be NULL.

Remarks

The IP is stored in network byte order, and the port is stored in host byte order. Returns false if there was an error parsing the string, or if a supplied hostname can't be resolved.

Possible string forms:

NULL => all IPs, any port (localAddress only).

"" => all IPs, any port (localAddress only).

"1.2.3.4" => 1.2.3.4 IP, any port (localAddress only).

"host.com" => host.com's IP, any port (localAddress only).

":2786" => all IPs, 2786 port (localAddress only).

"1.2.3.4:0" => 1.2.3.4 IP, any port (localAddress only).

"host.com:0" => host.com's IP, any port (localAddress only).

"0.0.0.0:2786" => all IPs, 2786 port (localAddress only).

"1.2.3.4:2786" => 1.2.3.4 IP, 2786 port (localAddress or remoteAddress).

"host.com:2786" => host.com's IP, 2786 port (localAddress or remoteAddress).

If this function needs to resolve a hostname ("host.com") it may need to contact a DNS server, which can cause the function to block for an indefinite period of time. Usually it is less than 2 seconds, but on certain systems, and under certain circumstances, it can take 30 seconds or longer.

Section Reference: [Gamespy Transport SDK](#)

See Also: [gt2Connect](#), [gt2CreateSocket](#), [gt2StringToHostInfo](#)

gt2StringToAliases

Get the aliases associated with a hostname or dotted IP.

```
char ** gt2StringToAliases(  
    const char * string );
```

Routine	Required Header	Distribution
gt2StringToAliases	<gt2.h>	SDKZIP

Return Value

Aliases associated with a hostname or dotted IP.

Parameters

string

[in] Hostname or IP to lookup.

Remarks

This is a utility function which provides a subset of the functionality of `gt2StringToHostInfo`. See the `gt2StringToHostInfo` documentation for important details.

Section Reference: [Gamespy Transport SDK](#)

See Also: [gt2StringToHostInfo](#)

gt2StringToHostInfo

Looks up DNS host information based on a hostname or dotted IP.

```
const char * gt2StringToHostInfo(  
    const char * string,  
    char *** aliases,  
    unsigned int *** ips );
```

Routine	Required Header	Distribution
gt2StringToHostInfo	<gt2.h>	SDKZIP

Return Value

The hostname associated with the string, or NULL if no information was available for the host.

Parameters

string

[in] Hostname ("www.gamespy.net") or dotted IP ("1.2.3.4") to lookup.

aliases

[in] On success, the variable passed in will point to a NULL-terminated list of alternate names for the host. Can be NULL.

ips

[in] On success, the variable passed in will point to a NULL-terminated list of alternate IPs for the host. Can be NULL.

Remarks

If the function can successfully lookup the host's info, the host's main hostname will be returned. If it cannot find the host's info, it returns NULL. For the aliases parameter, pass in a pointer to a variable of type (char **). If this parameter is not NULL, and the function succeeds, the variable will point to a NULL-terminated list of alternate names for the host.

For the ips parameter, pass in a pointer to a variable of type (int **). If this parameter is not NULL, and the function succeeds, the variable will point to a NULL-terminated list of alternate IPs for the host. Each element in the list is actually a pointer to an unsigned int, which is an IP address in network byte order.

The return value, aliases, and IPs all point to an internal data structure, and none of these values should be modified directly. Also, the data is only valid until another call needs to use the same data structure (virtually every internet address function will use this data structure). If the data will be needed in the future, it should be copied off.

This function may need to contact a DNS server, which can cause the function to block for an indefinite period of time. Usually it is < 2 seconds, but on certain systems, and under certain circumstances, it can take 30 seconds or longer.

Section Reference: [Gamespy Transport SDK](#)

See Also: [gt2IPToHostInfo](#), [gt2StringToHostname](#), [gt2StringToAliases](#), [gt2StringToIPs](#)

gt2StringToHostname

Get the hostname associated with a hostname or dotted IP.

```
const char * gt2StringToHostname(  
    const char * string );
```

Routine	Required Header	Distribution
gt2StringToHostname	<gt2.h>	SDKZIP

Return Value

Hostname associated with a hostname or dotted IP.

Parameters

string

[in] Hostname or IP to lookup.

Remarks

This is a utility function which provides a subset of the functionality of `gt2StringToHostInfo`. See the `gt2StringToHostInfo` documentation for important details.

Section Reference: [Gamespy Transport SDK](#)

See Also: [gt2StringToHostInfo](#)

gt2StringToIPs

Get the IPs associated with a hostname or dotted IP.

```
unsigned int ** gt2StringToIPs(  
    const char * string );
```

Routine	Required Header	Distribution
gt2StringToIPs	<gt2.h>	SDKZIP

Return Value

IPs associated with a hostname or dotted IP.

Parameters

string

[in] Hostname or IP to lookup.

Remarks

This is a utility function which provides a subset of the functionality of `gt2StringToHostInfo`. See the `gt2StringToHostInfo` documentation for important details.

Section Reference: [Gamespy Transport SDK](#)

See Also: [gt2StringToHostInfo](#)

gt2Think

Does any thinking for this socket and its connections.

```
void gt2Think(  
    GT2Socket socket );
```

Routine	Required Header	Distribution
gt2Think	<gt2.h>	SDKZIP

Parameters

socket

[in] The handle to the socket.

Remarks

Callbacks are typically called from within this function (although they can also be called from other places). It is possible that during this think the socket or any of its connections may be closed, so care must be taken if calling other GT2 functions immediately after thinking. The more frequently this function is called, the faster GT2 will be able to respond (and reply to) messages. The general rule is to call it as frequently as you can, although calling it faster than every 10-20 milliseconds is probably unnecessary. If you are using gt2Ping to measure ping times, then the accuracy of the latency measurement will increase with the frequency at which this function is called.

Section Reference: [Gamespy Transport SDK](#)

Transport SDK Callbacks

gt2ClosedCallback	This callback is called when the connection has been closed.
gt2ConnectAttemptCallback	This notifies the socket that a remote system is attempting a connection.
gt2ConnectedCallback	This callback is called when a connection attempt with gt2Connect finishes.
gt2DumpCallback	Called whenever data is sent or received over a socket.
gt2PingCallback	This callback is called when a response to a ping sent on this connection is received.
gt2ReceivedCallback	This callback is called when a message is sent from the remote system with a gt2Send.
gt2ReceiveFilterCallback	Callback for filtering incoming data.
gt2SendFilterCallback	Callback for filtering outgoing data.
gt2SocketErrorCallback	This callback is used to notify the application of a closed socket or fatal socket error condition.

[gt2UnrecognizedMessageCallback](#)

This callback gets called when the sock receives a message that it cannot match to an existing connection.

gt2ClosedCallback

This callback is called when the connection has been closed.

```
typedef void (*gt2ClosedCallback)(  
    GT2Connection connection,  
    GT2CloseReason reason );
```

Routine	Required Header	Distribution
gt2ClosedCallback	<gt2.h>	SDKZIP

Parameters

connection

[in] The handle to the connection.

reason

[in] The reason the connection closed.

Remarks

A connection close can be caused by either side calling `gt2CloseConnection` (or `gt2CloseConnectionHard`), either side closing the socket, or some sort of error. The connection cannot be used again once this callback returns.

Section Reference: [Gamespy Transport SDK](#)

See Also: [gt2CloseConnection](#), [gt2CloseConnectionHard](#), [gt2Connect](#), [gt2Accept](#)

gt2ConnectAttemptCallback

This notifies the socket that a remote system is attempting a connection.

```
typedef void (*gt2ConnectAttemptCallback)(  
    GT2Socket socket,  
    GT2Connection connection,  
    unsigned int ip,  
    unsigned short port,  
    int latency,  
    GT2Byte * message,  
    int len );
```

Routine	Required Header	Distribution
gt2ConnectAttemptCallback	<gt2.h>	SDKZIP

Parameters

socket

[in] The handle to the socket.

connection

[in] The handle to the connection.

ip

[in] The IP (network byte order) from which the connect attempt is coming.

port

[in] The port (host byte order) from which the connect attempt is coming

latency

[in] estimate of the round-trip time between the two machines (in milliseconds).

message

[in] Optional initial data sent with the connect attempt. May be NULL.

len

[in] Length of the initial data. May be 0.

Remarks

The IP and port of the remote system is provided, along with an optional initial message, and a latency estimate. These can be used to validate/authenticate the connecting system. This connection must either be accepted with `gt2Accept`, or rejected with `gt2Reject`. These can be called from within this callback, however they do not need to be. They can be called at any time after this callback is received. This is very useful for systems that need to check with another machine to authenticate the user (such as for a CDKey system). The latency is only an estimate, however it can be used for things such as only allowing low-ping or high-ping users onto a server.

Section Reference: [Gamespy Transport SDK](#)

See Also: [gt2Listen](#), [gt2Connect](#), [gt2Accept](#), [gt2Reject](#)

gt2ConnectedCallback

This callback is called when a connection attempt with gt2Connect finishes.

```
typedef void (*gt2ConnectedCallback)(  
    GT2Connection connection,  
    GT2Result result,  
    GT2Byte * message,  
    int len );
```

Routine	Required Header	Distribution
gt2ConnectedCallback	<gt2.h>	SDKZIP

Parameters

connection

[in] The handle to the connection.

result

[in] The result of the connection attempt. Anything aside from GT2Success indicates failure.

message

[in] If result is GT2Rejected, this is the rejection message. May be NULL.

len

[in] If result is GT2Rejected, the length of message. May be 0.

Remarks

If result is GT2Success, then this connection attempt succeeded. The connection object can now be used for sending/receiving messages. Any other result indicates connection failure, and the connection object cannot be used again after this callback returns. If the result is GT2Rejected, then message contains an optional rejection message sent by the listener. If result is not GT2Rejected, then message will be NULL and len will be 0.

Section Reference: [Gamespy Transport SDK](#)

See Also: [gt2Connect](#)

gt2DumpCallback

Called whenever data is sent or received over a socket.

```
typedef void (*gt2DumpCallback)(  
    GT2Socket socket,  
    GT2Connection connection,  
    unsigned long ip,  
    unsigned short port,  
    GT2Bool reset,  
    const GT2Byte * message,  
    int len );
```

Routine	Required Header	Distribution
gt2DumpCallback	<gt2.h>	SDKZIP

Parameters

socket

[in] The handle to the socket.

connection

[in] The handle to the connection associated with this message, or NULL if there is no connection for this message.

ip

[in] The remote IP address, in network byte order.

port

[in] The remote host, in host byte order.

reset

[in] If true, the connection has been reset (only used by the receive callback).

message

[in] The message (should not be modified).

len

[in] The length of the message.

Remarks

Trying to send a message from within the send dump callback, or letting the socket think from within the receive dump callback can cause serious problems, and should not be done.

Section Reference: [Gamespy Transport SDK](#)

See Also: [gt2SetSendDump](#), [gt2SetReceiveDump](#)

gt2PingCallback

This callback is called when a response to a ping sent on this connection is received.

```
typedef void (*gt2PingCallback)(  
    GT2Connection connection,  
    int latency );
```

Routine	Required Header	Distribution
gt2PingCallback	<gt2.h>	SDKZIP

Parameters

connection

[in] The handle to the connection.

latency

[in] The round trip time of the ping, in milliseconds.

Remarks

This callback gives a measure of the time it takes for a datagram to make a round-trip from one connection to the other. The latency reported in this callback will typically be larger than that reported by using ICMP pings between the two machines (the "ping" program uses ICMP pings), because ICMP pings happen at a lower level in the operating system. However, the ping reported in this callback will much more accurately reflect the latency of the application, as the application's messages must go through the same path as these pings, as opposed to ICMP. Because pings are unreliable, a ping sent with gt2Ping is not guaranteed to make it through the entire round-trip. So not every call to gt2Ping will result in this callback being called. In addition, unreliable messages may be repeated (although this is a very rare occurrence), which means this callback could be called multiple times for a single call to gt2Ping.

Section Reference: [Gamespy Transport SDK](#)

See Also: [gt2Ping](#)

gt2ReceivedCallback

This callback is called when a message is sent from the remote system with a gt2Send.

```
typedef void (*gt2ReceivedCallback)(  
    GT2Connection connection,  
    GT2Byte * message,  
    int len,  
    GT2Bool reliable );
```

Routine	Required Header	Distribution
gt2ReceivedCallback	<gt2.h>	SDKZIP

Parameters

connection

[in] The handle to the connection.

message

[in] The message that was sent. May be NULL.

len

[in] The length of the message. May be 0.

reliable

[in] GT2True if the message was sent reliably.

Remarks

If an message is sent from the remote end of the connection reliably, then it will always be received with this callback. If it is not sent reliably, then the message might not be received, it might be received out of order, or it might be received more than once (very rare).

Section Reference: [Gamespy Transport SDK](#)

See Also: [gt2Send](#)

gt2ReceiveFilterCallback

Callback for filtering incoming data.

```
typedef void (*gt2ReceiveFilterCallback)(  
    GT2Connection connection,  
    int filterID,  
    GT2Byte * message,  
    int len,  
    GT2Bool reliable );
```

Routine	Required Header	Distribution
gt2ReceiveFilterCallback	<gt2.h>	SDKZIP

Parameters

connection

[in] The handle to the connection.

filterID

[in] Pass this ID to `gtFilteredReceive`.

message

[in] The message that was received. Will be NULL if an empty message.

len

[in] The length of the message in bytes. Will be 0 if an empty message.

reliable

[in] True if this is a reliable message.

Remarks

Call `gt2FilteredReceive` with the filtered data, either from within the callback or later.

The message may point to a memory location supplied to `gt2FilteredReceive` by a previous filter, so if this filter's call to `gt2FilteredReceive` is delayed, it is the filter's responsibility to make sure the data is still around when and if it is needed.

Section Reference: [Gamespy Transport SDK](#)

See Also: [gt2AddReceiveFilter](#), [gt2RemoveReceiveFilter](#), [gt2FilteredReceive](#)

gt2SendFilterCallback

Callback for filtering outgoing data.

```
typedef void (*gt2SendFilterCallback)(  
    GT2Connection connection,  
    int filterID,  
    const GT2Byte * message,  
    int len,  
    GT2Bool reliable );
```

Routine	Required Header	Distribution
gt2SendFilterCallback	<gt2.h>	SDKZIP

Parameters

connection

[in] The handle to the connection.

filterID

[in] Pass this ID to `gt2FilteredSend`.

message

[in] The message being sent. Will be NULL if an empty message.

len

[in] The length of the message being sent, in bytes. Will be 0 if an empty message.

reliable

[in] If the message is being sent reliably.

Remarks

Call `gt2FilteredSend` with the filtered data, either from within the callback or later.

The message points to the same memory location as the message passed to `gt2Send` (or `gt2FilteredSend`), so if the call to `gt2FilteredSend` is delayed, it is the filter's responsibility to make sure the data is still around when and if it is needed.

Section Reference: [Gamespy Transport SDK](#)

See Also: [gt2AddSendFilter](#), [gt2RemoveSendFilter](#), [gt2FilteredSend](#)

gt2SocketErrorCallback

This callback is used to notify the application of a closed socket or fatal socket error condition.

```
typedef void (*gt2SocketErrorCallback)(  
    GT2Socket socket );
```

Routine	Required Header	Distribution
gt2SocketErrorCallback	<gt2.h>	SDKZIP

Parameters

socket

[in] The handle to the socket.

Remarks

Once this callback returns, the socket and all of its connections are invalid and can no longer be used. All connections that use this socket are terminated, and their `gt2CloseCallback` callbacks will be called before this callback is called (with the reason set to `GT2SocketError`).

Section Reference: [Gamespy Transport SDK](#)

See Also: [gt2CreateSocket](#)

gt2UnrecognizedMessageCallback

This callback gets called when the sock receives a message that it cannot match to an existing connection.

```
typedef GT2Bool (*gt2UnrecognizedMessageCallback)(  
    GT2Socket socket,  
    unsigned int ip,  
    unsigned short port,  
    GT2Byte * message,  
    int len );
```

Routine	Required Header	Distribution
gt2UnrecognizedMessageCallback	<gt2.h>	SDKZIP

Return Value

GT2True if the callback recognizes the message and handles it.

GT2False if GT2 should handle the message.

Parameters

socket

[in] The handle to the socket.

ip

[in] The IP address of the remote machine the message came from (in network byte order).

port

[in] The port on the remote machine (in host byte order).

message

[in] The message (may be NULL for an empty message).

len

[in] The length of the message (may be 0).

Remarks

If the callback recognizes the message and handles it, it should return GT2True, which will tell the socket to ignore the message. If the callback does not recognize the message, it should return GT2False, which tells the socket to let the other side know there is no connection.

Section Reference: [Gamespy Transport SDK](#)

See Also: [gt2SetUnrecognizedMessageCallback](#), [gt2GetSocketSOCKET](#)

Transport SDK Structures

[GT2ConnectionCallbacks](#)

Callbacks set for each connection.

GT2ConnectionCallbacks

Callbacks set for each connection.

```
typedef struct  
{  
    gt2ConnectedCallback connected;  
    gt2ReceivedCallback received;  
    gt2ClosedCallback closed;  
    gt2PingCallback ping;  
} GT2ConnectionCallbacks;
```

Members

connected

Called when gt2Connect is complete.

received

Called when a message is received.

closed

Called when the connection is closed (remotely or locally).

ping

Called when a ping reply is received.

Section Reference: [Gamespy Transport SDK](#)

Transport SDK Enumerations

GT2CloseReason	Reason the connection was closed.
GT2ConnectionState	Possible states for any GT2Connection.
GT2Result	Result of a GT2 operation. Check individual function definitions to see possible results.

GT2CloseReason

Reason the connection was closed.

```
typedef enum  
{  
    GT2LocalClose,  
    GT2RemoteClose,  
    GT2CommunicationError,  
    GT2SocketError,  
    GT2NotEnoughMemory  
} GT2CloseReason;
```

Constants

GT2LocalClose

The connection was closed with `gt2CloseConnection`.

GT2RemoteClose

The connection was closed remotely.

GT2CommunicationError

An invalid message was received (it was either unexpected or wrongly formatted).

GT2SocketError

An error with the socket forced the connection to close.

GT2NotEnoughMemory

There wasn't enough memory to store an incoming or outgoing message.

Section Reference: [Gamespy Transport SDK](#)

GT2ConnectionState

Possible states for any GT2Connection.

```
typedef enum  
{  
    GT2Connecting,  
    GT2Connected,  
    GT2Closing,  
    GT2Closed  
} GT2ConnectionState;
```

Constants

GT2Connecting

Negotiating the connection.

GT2Connected

Connection is active.

GT2Closing

Connection is being closed.

GT2Closed

Connection has been closed and can no longer be used.

Section Reference: [Gamespy Transport SDK](#)

GT2Result

Result of a GT2 operation. Check individual function definitions to see possible results.

typedef enum

```
{  
    GT2Success,  
    GT2OutOfMemory,  
    GT2Rejected,  
    GT2NetworkError,  
    GT2AddressError,  
    GT2DuplicateAddress,  
    GT2TimedOut,  
    GT2NegotiationError  
} GT2Result;
```

Constants

GT2Success

Success.

GT2OutOfMemory

Ran out of memory.

GT2Rejected

Attempt rejected.

GT2NetworkError

Networking error (could be local or remote).

GT2AddressError

Invalid or unreachable address.

GT2DuplicateAddress

A connection was attempted to an address that already has a connection on the socket.

GT2TimedOut

Timeout reached.

GT2NegotiationError

Error negotiating with the remote side.

Section Reference: [Gamespy Transport SDK](#)

Voice 2 SDK

Overview

The GameSpy Voice SDK 2 (GV) is a library that facilitates in-game voice communication between players. Someone with little or no voice experience can easily use GV without having to learn about the details of voice capture, playback, or encoding.

GV is simple enough to be easily and quickly added to an application, while also being powerful and flexible enough to fit within virtually any networking architecture. GV is also extremely efficient in its use of memory, bandwidth, and processor time.

GV delivers optimal performance, in a simple API, making it as easy as possible to add to any application.

Much of the power of GV comes from the fact that it does not impose any networking constraints on the application. It captures audio, encodes it, then passes it to the application. The application is responsible for routing the encoded audio to its final destination(s), most likely over its existing game networking. At the destination, the application gives the encoded audio back to GV, which then decodes it, mixes it into an audio stream, and plays it.

GV is written in standard ANSI C and can be used for Win32 or PS2 applications. Dedicated servers can be run on any platform. Just include all of the source files in your project, and you can start talking over the Internet.

GV relies on system-specific hardware libraries to interface with audio devices, and it uses freely available codec (compression and decompression) libraries to handle the actual voice compression. See the Requirements section of this document for further information on these libraries.

The SDK package also includes a test app and a sample app. The test app shows a basic C implementation of the SDK. It will help in

understand the SDK's core functionality and it will also provide you with a baseline implementation for testing the SDK on your target system. The sample app is written using Win32 and MFC. It shows a sample graphical implementation of the SDK. See the Samples section of this document for further information on the sample and test apps.

The rest of this document presents a simple set of instructions for using GV. See the reference documentation for further details.

File Manifest

The following files should be included with this package. If any of the files are missing, please [contact devsupport@gamespy.com](mailto:devsupport@gamespy.com).

File	Description
<i>gv.h</i>	GV header (all user functions are prototyped here)
<i>gvCodec.c,h</i>	Encryption and decryption of audio frames
<i>gvCustomerDevice.c,h</i>	Custom capture/playback device interface
<i>gvDevice.c,h</i>	Device interface
<i>gvDirectSound.c,h</i>	DirectSound interface (only used with Win32)
<i>gvFrame.c,h</i>	Common code for handling audio frames
<i>gvLogitechPS2Codecs.c,h</i>	Interface to Logitech's PS2 voice codec library (only used with PS2)
<i>gvMain.c,h</i>	The main entry point for most GV functionality
<i>gvPS2Audio.c,h</i> (only used with PS2)	Interfaces with hardware-specific PS2 libraries
<i>gvPS2Eyeto.c,h</i>	Eyeto interface (only used with PS2)
<i>gvPS2Headset.c,h</i>	USB audio device (headset/microphone/speakers) interface (only used with PS2)
<i>gvPS2Spu2.c,h</i> with PS2)	SPU2 (system/TV audio) interface (only used with PS2)
<i>gvSource.c,h</i>	Common code for handling audio sources
<i>gvSpeex.c,h</i> with Win32)	Interface to the Speex codec library (only used with Win32)
<i>gvUtil.c,h</i>	Miscellaneous common code
<i>nonport.c,h</i>	Platform-specific code
<i>darray.c,h</i>	Code for managing dynamic arrays
<i>/Voice2Test/</i>	ANSI-C sample

/Voice2BuddyMFC/ Windows MFC sample

Requirements

GV relies on a few other libraries provided by both GameSpy and 3rd-parties. The 3rd party libraries are all freely available. See below for further information, depending on your platform.

Note that you can use custom hardware devices and/or custom codecs instead of those provided by default. If doing so, you will not need the corresponding 3rd-party libraries.

Common Code

GV uses the GameSpy Common Code package, the latest version of which is available from <http://www.gamespy.net>. The Voice 2 zip and the Common Code zip should both be extracted into the same directory. This will result in the GV files being in a Voice2 subdirectory, and the common code will be in the directory's root.

Win32

GV supports the Win32 platform by using the DirectSound library (part of DirectX) for interfacing with voice capture and playback devices and the Speex library for compression and decompression of audio.

DirectX

The latest version of DirectX can be downloaded from <http://msdn.microsoft.com/downloads/>. The site also has information on redistributing DirectX with your application. GV was developed with DirectX 9.0. Though it should work with previous versions it is highly recommended that you use the latest version of DirectX. Once DirectX has been installed, ensure that its include folder is in your include path and its lib folder is in your link path. You will also need to setup any application that uses GV to link with dsound.lib and dxguid.lib.

Speex

Speex is a freely available, open source audio codec (<http://speex.org/>).

Speex licensing information is available from <http://www.xiph.org/licenses/bsd/speex/>.

Please review the licensing information carefully before using the Speex library.

GV currently uses version 1.0.5 of Speex, which can be downloaded from <http://downloads.us.xiph.org/releases/speex/speex-1.0.5.tar.gz>. In order for the sample and test applications to function, you will need to extract the tar into the Voice2 directory. This will create a speex-1.0.5 subdirectory in the Voice2 directory. The SDK itself does not rely on Speex being in that directory, so you can put it anywhere you would like for your own application. However, wherever you put it, its include folder must be in your include path. You will also need to add all of the source files in the libspeex directory to your project, with the exception of any test*.c files.

When compiling the Speex libraries at the default MSVC++ warning level, there are a number of warnings due to missing explicit casts. It is fairly easy to disable these warnings using the following line:

```
#pragma warning ( disable : 4244 4305 4100 4127 )
```

To disable the warnings, put the line in all of the files which generate the warnings, or in a header which can then be easily included in any files which generate the warnings.

PS2

GV supports PS2 devices through three libraries provided by Logitech specifically for the PS2. IgAud is used to interface with USB audio devices, IgVid is used to interface with the microphone on EyeToy devices, and IgCodec is used to compress captured audio. IgAud and IgCodec are available at <https://www.ps2-pro.com/projects/lgaud/>, and IgVid is available at <https://www.ps2-pro.com/projects/liblgvid/>. SPU2 output (direct system output, generally through the TV) is also supported, allowing users without headsets or USB speakers to hear other users.

PS2 applications are required to load certain IRX modules to use the

various hardware devices. These are lgaud.irx and usbd.irx for IgAud, lgvid.irx for IgVid, and libsd.irx and sdrdrv.irx for SPU2. Sample loading code is available in load_voice_modules() in ps2common.c. Applications must also link with liblgaud.a for IgAud, liblgvid.a for IgVid, and libscf.a and libsdr.a for SPU2. To exclude support for particular hardware device types, see the Advanced section of this document.

IgAud

GV was developed with version 1.10.001 of IgAud. A license agreement for IgAud, liblgaud_license_agreement.pdf, is in the tar.

Please read the licensing agreement carefully before using the library.

See the ReadMe.1st in the tar for information on installing IgAud. The IgAud documentation provides details on lgaud.irx loading parameters.

IgVid

GV was developed with version 1.07.008 of IgVid. The license agreement for IgVid can be found at the end of lgvid.pdf, which can be found in the tar's doc directory. GV now supports version 2.01.003. If you wish to support the older library, please use the following preprocessor directive in your project: [GVI_LGVID_OLD_DRIVER](#)

Please read the licensing agreement carefully before using the library.

See the README.1st in the tar for information on installing IgVid. The IgVid documentation provides details on lgvid.irx loading parameters.

IgCodec

GV was developed with version 1.00.002 of IgCodec. See the README in the tar for information on installing IgCodec. The bottom of the README also contains a "TERMS OF USE" section which you should read carefully before using the library. Any program that uses GV will

need to link with liblgcodec.lib.

Note that because the PS2 uses a different codec library than the Win32 version of GV, users on the two different platforms cannot talk with each other. However they can be made compatible by using a custom codec.

PS3

Currently the SDK supports any PS3 compatible USB/Bluetooth Headset. Future versions will have support for other devices like the Eye Toy. Speex is currently the only codec supported on the PS3. Refer to the instructions in the Win32 section for obtaining speex.

Linux, Mac, etc.

Only Win32 and PS2 hardware devices are included in GV at present. However, because applications provide their own routing between users, dedicated servers do not need to use the GV SDK, which allows them to be run on any platform, such as Linux.

Also, because GV supports custom devices and codecs, clients can be written for other platforms as well, as long as the application provides the interface to the hardware and the codec.

Sample

The sample app included with this SDK, *Voice2BuddyMFC*, uses a few of the other GameSpy SDKs. These are GameSpy Transport 2 (GT2), NAT Negotiation, and GameSpy Presence (GP), which are all included under the same license as GV. If you do not have these SDKs, you can download them from <http://gamespy.net/secure/download/>. Extract them into the same root as the common code, just as you did with the GV code.

Samples

The GV package includes a test app and a sample app. The test app, *Voice2Test*, is a very simple implementation of the SDK, while the sample app, *Voice2BuddyMFC*, is designed to show a graphical implementation of the SDK.

Voice2Test

The test app is written in ANSI C, and it can be used as a simple test to make sure GV works on your system. It takes as its only argument a dotted IP address, to which it will send all captured voice packets. It also opens a socket on a fixed port and will play any voice packets it receives on the socket. The app will run until either Q is pressed on the PC, or the X is pressed on the PS2. To stop capture use V or the circle button, and to start capture again use C or the square button.

There are a set of defines at the top of the source file, *Voice2Test.c*, which can be used to change various aspects of the test app. `LOCAL_ECHO` can be set to 1 to turn local on (you will hear your own voice). `RAW_CODEC` can be set to 1 to switch on the use of a custom codec which does not actually do any compression (but can be easily modified to test a codec). `SHOW_TALKERS` can be set to 1 to have the app print out the list of talkers once a second. `SHOW_VOLUME` can be set to 1 to have the app print out the volume of all captured audio packets. `CAPTURE_THRESHOLD` sets the capture threshold. A value of 0.0 means there will be no threshold, and a value greater than 0.0 and less than or equal to 1.0 will set the threshold to that value.

Voice2BuddyMFC

The sample app is written with MFC and is designed to show a graphical implementation of GV. This sample app may be used to establish voice communication with a single member of your buddy list. (Your game, of course, is not limited to voice communication with a single player.)

Voice2BuddyMFC makes use of three other GameSpy SDKs.

Presence SDK

Used to retrieve buddy list.

Nat Negotiation SDK

Used to connect to buddies.

Transport 2 SDK

Used to manage connections to buddies and voice data transmission.

Note: These SDKs are not required to use GV. They are used in the sample application to simplify the matchmaking process. In a typical game a networking channel has already been established between players (or from host to player). Voice data may be sent using the network channel used for game data.

The *Voice2Buddy* sample has three major stages. These stages are Setup, Login and VoiceSession.

Setup

The Setup stage consists of a dialog in which the user may select input and output devices as well as set the voice activation level. It is recommend that you expose a similar dialog in your application, as the user may have multiple input and output devices from which to select (e.g. a USB headset for voice communication and speakers connected to the sound card for game audio). DirectX 9 has a standard dialog which may be displayed.

Login

The Login dialog contains the standard fields used to log into the GameSpy ID system. (Email, nickname and password.) After logging in, the user will be presented with a list of buddies that they may invite to a voice session. The buddy may then accept the invitation which will result in the creation of the VoiceSession.

VoiceSession

The third stage, VoiceSession, is not a stage within the SDK. GV does not require the creation of a "channel" or "server". The receiving application simply decides, "Do I want to play Voice data from this person?".

In the sample, all voice data is immediately presented to the SDK for playback. If you wish to mute a user, simply discard the voice data instead of passing it into GV.

Terms and Concepts

This section will explain some of the concepts and terminology used by GV. Reading this section will greatly assist in your ability to understand the rest of this document.

Audio

Samples and Rates

One of the primary functions of GV is to capture and playback audio. Digital audio consists of a set of samples captured at a uniform rate. The samples form a waveform - in other words, each sample is the value of the waveform at a particular instance in time. The number of bits used to store the value of each sample determines the bit rate, and the number of samples stored in one second determines the sample rate. GV uses a sample rate of 8000 samples per second and a bit rate of 16 bits per sample, which are de facto standards for Internet voice applications. These rates are defined at the top of gv.h. Also, GV has a [GVSample](#) type which represents a single sample as a signed short.

Frames

GV typically deals with audio in units called frames. A frame of audio will always represent the same number of samples, and so it will always represent the same amount of time. Typically frames represent 160 samples, which is 20ms at the standard sample rate (160 samples / 8000 samples per second = 0.02 seconds = 20ms). The exact number of samples per frame depends on the codec, but almost all codecs use 160 samples per frame.

A frame can either be raw or encoded. A raw frame is an array of GVSamples which will be, at the standard frame size and bit rate, 320 bytes. The size, however, can vary with codecs that use other frame sizes. Raw frames are used with custom devices, custom codecs, and filters, as they need to deal with the raw, uncompressed audio. An encoded frame is a raw frame that has been compressed by the codec. It

is stored in an array of bytes, using the `GVByte` type (an unsigned char). The size of a compressed frame depends on the codec and can vary from less than 10 bytes to 300 or more bytes, although it will typically be in the range of about 15 to 35 bytes. Encoded frames are used with custom codecs, audio capture, and audio playback. GV provides functionality for obtaining the current codec's samples per frame (number of samples in a raw frame) and encoded frame size (number of bytes in an encoded frame).

Packets

A packet represents a set of one or more encoded frames stored in a contiguous block of memory (an array of `GVBytes`). Therefore the length of a packet will always be a multiple of the size of an encoded frame. When GV passes the application audio that it has captured and compressed, it does so in the form of a packet. This packet is ready to be sent over the Internet (along with some other meta-data described below). When it is received by another player, it can be passed directly to the SDK (along with the meta-data) to be played.

Frame Stamps

A frame stamp stores a value that GV uses to synchronize packets for playback. When a packet is captured, GV will pass a frame stamp to the application along with the packet. The frame stamp marks the time at which the first frame in the packet was captured. If the packet is sent over the Internet, the frame stamp should be sent along with it. On playback, the frame stamp must be passed back to GV along with the packet. A frame stamp is represented by the `GVFrameStamp` type, which is 2 bytes.

Sources

Sources are used by GV to uniquely identify users, which enables it to synchronize multiple incoming streams of audio. Whenever a packet is passed from the application to GV to be played, a source must be passed along with the packet. This allows GV to identify who spoke that particular packet. A source is represented by the `GVSource` type. By

default this is an int, however the `GV_CUSTOM_SOURCE_TYPE` define can be used to have a `GVSource` represent any arbitrary type, such as a `sockaddr_in` (Internet IP and port). The source can be a player's index, connection ID, Internet address, or any data that uniquely identifies a talker. The source may need to be sent along with the packet.

One example of this would be a client-server application in which all of the packets pass through a server before being sent to their final destination. For a peer-to-peer game, however, in which packets are received directly from the player who generated them, the connection on which the packets were received could be used to identify the source.

Local Echo

Local echo is what happens when you locally playback any captured packets. It results in the user being able to hear himself talking, although there is a slight delay due to the fact that the audio is being processed by both GV and the application. Local echo is very simple with GV - just pass captured packets directly to a playback device.

Devices

A device is a particular piece of hardware on the system that can be used by GV to capture and/or playback audio. On some platforms, such as Win32, a device will only do one thing - capture or playback. Even though a sound card may be one physical piece of hardware, it is represented through GV as two separate devices, one which handles capture and one which handles playback. This is a consequence of how DirectSound works. On other platforms, such as the PS2, a single device may do both capture and playback, although it is still possible that it can only do one or the other. For example headsets will support both capture and playback, while EyeToy cameras can only be used for capture.

GV supports mixing and matching of devices. For example, a USB headset could be used for voice capture, a sound card outputting to speakers could be used for general game audio, and the headset could be used for voice communication from other players. There is nothing preventing one or more devices from being used for capture and a different set of one or more devices from being for playback. These

devices can be a mix of devices supported internally and custom devices.

Device IDs

A device is uniquely identified by a [GVDeviceID](#). This type is represented by a GUID on Win32 and by an int on the PS2. GV can provide you with a list of devices available on a particular system, and each device will have its own [GVDeviceID](#). When you initialize a device through GV, you use the GVDeviceID to tell GV which device you are attempting to initialize.

Devices

When the application initializes a device through GV, it obtains a reference to the device as a GVDevice value. The application will need to use the [GVDevice](#) handle whenever it does anything related to the device. GV will manage the device until the application frees it (through GV). After it has been freed the [GVDevice](#) handle will no longer be valid and can be set to NULL, which will never represent a valid GVDevice.

Device Types

Any particular device is capable of capture, playback, or both. When an operation needs to specify that it applies to only capture, only playback, or both capture and playback, it uses the [GVDeviceType](#) type to specify. The typical values are [GV_CAPTURE](#), [GV_PLAYBACK](#), and [GV_CAPTURE_AND_PLAYBACK](#). [GV_CAPTURE](#) and [GV_PLAYBACK](#) are bitfields, so [GV_CAPTURE_AND_PLAYBACK](#) is equal to $(GV_CAPTURE|GV_PLAYBACK)$. If a particular [GVDeviceType](#) value is 0, that means that it does not apply to capture or playback. When the available devices are listed, there is a [GVDeviceType](#) for each field that specifies if it is the system default for capture, playback, both, or neither. If it is neither, then its value is 0.

Custom Devices

GV supports the use of custom devices. This allows you to write your

own audio device interface. You can use this to, for example, pass the playback audio through your own internal audio system instead of directly to the sound card. See the Advanced section of this document for further information.

Codecs

GV uses codecs to compress and decompress audio.

Codecs

There is a set of default codecs available on each system, represented by the [GVCodec](#) type. These range from codecs that preserve audio quality at the expense of greater bandwidth usage to lower audio quality codecs that use less bandwidth. GV does not support the use of multiple codecs - after GV is initialized, but before any devices are initialized, the application must set the codec to be used. The codec can then not be changed once devices have been initialized.

GV can only handle packets that were compressed using the codec it has been set to use. It is the applications responsibility to ensure that GV is not given packets that were compressed using another method.

Custom Codecs

GV supports the use of a custom codec, which allows developers to use any desired codec with GV. See the Advanced section of this document for further information.

Implementation

This section will explain all of the basic operations that you can perform using GV. After reading this section you should be able to write an application that can capture and playback voice audio.

Initialization and Cleanup

Before doing anything else with GV, you must call `gvStartup`. On Win32 the prototype is:

```
GVBool gvStartup(HWND hwnd);
```

On other platforms the prototype is:

```
GVBool gvStartup(void);
```

The function does any necessary internal initialization. It will return `GVFalse` in case of an error initializing. The `HWND` passed to the Win32 version is the handle for the application's main window. This can be `NULL` if the application does not have a main window.

To cleanup the SDK, use `gvCleanup`:

```
void gvCleanup(void);
```

The function will do any necessary internal cleanup. GV cannot be used again until `gvStartup` is called.

Setting A Codec

The first thing to do after initializing the SDK is to set the codec you would like to use. The codec cannot be changed while any devices are initialized, so the codec an application will typically just set the codec once, when it starts using voice. For information on using a custom codec, see the Advanced section of this document.

```
GVBool gvSetCodec(GVCodec codec);
```

The codec must be one of the following values:

- [GVCodecSuperHighQuality](#)
- [GVCodecHighQuality](#)
- [GVCodecAverage](#)
- [GVCodecLowBandwidth](#)
- [GVCodecSuperLowBandwidth](#)

The codecs are arranged in order of descending quality and bandwidth. In other words, the codecs higher up on the list are of higher audio quality and use more bandwidth, while the codecs lower on the list are of lower audio quality and use less bandwidth.

The [GVCodecAverage](#) codec produces good quality audio with a reasonable bandwidth cost. It is generally the best codec to use, and you should only use another codec if you are restricted to lower bandwidth or need high quality audio.

The particular stats for a codec can be obtained using [gvGetCodecInfo](#).

```
void gvGetCodecInfo(int * samplesPerFrame, int * enc
```

This function returns the samples per frame, encoded frame size (in bytes), and bits per second for the currently selected codec. Note that the bits per second doesn't take into account any overhead, such as the need to transmit a frame stamp value with each packet. It is based only on the encoded frame size and the number of frames per second.

It is important that all users have the same codec set. If users attempt to communicate using different codecs, the result will most likely be unpredictable audio.

Listing Devices

The application uses `gvListDevices` to get a list of the devices available on the system.

```
int gvListDevices(GVDeviceInfo devices[], int maxDev
```

Pass in an array of `GVDeviceInfos`, which will be filled in by the function, the number of elements in the array, and the types of devices that you want to be listed. You can request capture devices with `GV_CAPTURE`, playback devices with `GV_PLAYBACK`, or capture and playback devices with `GV_CAPTURE_AND_PLAYBACK`. For `GV_CAPTURE_AND_PLAYBACK`, it can list capture devices, playback devices, and devices that support both capture and playback. The function will return the number of devices that it put in the list, which may be less than the value that was passed in for `maxDevices`. If 0 is returned, then either there was an error listing devices or no devices were found. For each device that is listed, a `GVDeviceInfo` will be filled in with details on the device.

```
typedef struct
{
    GVDeviceID m_id;
    char m_name[GV_DEVICE_NAME_LEN];
    GVDeviceType m_deviceType;
    GVDeviceType m_defaultDevice;    // not supp
    GVHardwareType m_hardwareType;
} GVDeviceInfo;
```

The `m_id` is used if you initialize this device with `gvNewDevice`. The `m_name` contains a user-readable name for the device. The `m_deviceType` indicates if this device is for capture, playback, or both capture and playback. The `m_defaultDevice` indicates if this device is the default capture device, default playback device, both, or neither. If neither, the value will be 0 (this will always be the case on the PS2, as it does not have a default device indicator). The `m_hardwareType` is used to give the application more information about the device's actual hardware. Under Win32 this will always be `GVHardwareDirectSound` (no further information is available). With the PS2 this will be

[GVHardwarePS2Spu2](#) for the SPU2 device, [GVHardwarePS2Headset](#) for USB headsets, [GVHardwarePS2Microphones](#) for USB microphones, [GVHardwarePS2Speakers](#) for USB speakers, or [GVHardwarePS2Eyeto](#)y for Eyeto devices. The SPU2 device will always be the first device listed on the PS2.

Note that on the PS2, there is a small delay between when the IRX modules are loaded and when USB devices are actually detected. This is why *Voice2Test* pauses for one second right before calling [gvListDevices](#). However in a real application this pause won't be necessary, as long as the IRX modules are loaded well before the user would get to a voice configuration screen.

Creating and Freeing Devices

To initialize a device, you use [gvNewDevice](#).

```
GVDevice gvNewDevice(GVDeviceID deviceID, GVDeviceTy
```

You pass the function the [GVDeviceID](#) for the device that you want to initialize. Also pass in a type, which will tell GV if you want to initialize the device for capture, playback, or both. A device that supports both capture and playback may be initialized for just one or the other (or both). If the device was successfully initialized, a handle to the device will be returned. If there was an error setting up the device, NULL will be returned.

With Win32, there are two globally defined default devices which you can use. The [GVDeviceIDs](#) for these are [GVDefaultCaptureDeviceID](#) and [GVDefaultPlaybackDeviceID](#). So, instead of calling [gvListDevices](#), you can simply use the defaults. However this is only recommended during development - for release your application should allow the user to choose which device to use. Also note that the default device IDs will not match the [GVDeviceIDs](#) of the default devices listed by [gvListDevices](#), although they will represent the same physical hardware.

With the PS2, the SPU2 device is globally defined as

[GVPS2Spu2DeviceID](#). This allows you to use the SPU2 device for output without having to list devices. Again, this is only recommend for use during development. For release the application should list devices and either allow the user to choose a device, or use the SPU2 if no other playback devices are found. Unlike the Win32 default devices, [GVPS2Spu2DeviceID](#) will match the [GVDeviceID](#) of the SPU2 device (always the first in the list).

When you are done using the device, use [gvFreeDevice](#) so that GV can clean it up.

```
void gvFreeDevice(GVDevice device);
```

Once a device has been freed it can no longer be used. After calling this function you should set the variable in which you stored the device handle to NULL.

Starting and Stopping Devices

Once a device has been initialized, it is ready to start capturing or playing audio. After a capture device is started, it will begin capturing audio and passing it back to the application. After a playback device is started, it will play any audio that the application passes it. To start a device, use [gvStartDevice](#).

```
GVBool gvStartDevice(GVDevice device, GVDeviceType t
```

The device parameter is the handle of the device to start. The type parameter specifies if the device should start capturing ([GV_CAPTURE](#)), playing ([GV_PLAYBACK](#)), or capturing and playing ([GV_CAPTURE_AND_PLAYBACK](#)). For devices that support both capture and playback, each can be started independently. The function will return [GVTrue](#) if the device was started successfully, and it will return [GVFalse](#) if there was an error.

When you want a device to stop capturing or playing, use [gvStopDevice](#).

```
void gvStopDevice(GVDevice device, GVDeviceType type)
```

When a capture device is stopped, it will stop passing captured audio to the application. When a playback device is stopped, it will stop playing audio. For devices that support both capture and playback, each can be stopped independently.

Use [gvIsDeviceStarted](#) to check if a device has been started or not.

```
GVBool gvIsDeviceStarted(GVDevice device, GVDeviceType type)
```

Capturing Packets

Once a capture device has been started, it will immediately start filling an internal buffer (which may or may not be on the actual sound hardware) with audio data. The application calls [gvCapturePacket](#) to take captured audio out of the buffer.

```
GVBool gvCapturePacket(GVDevice device, GVByte * packet, int len, GVByte * frameStamp, float volume)
```

The first parameter is a handle to the capture device. The packet parameter points to a block of memory that must be large enough to hold at least one encoded frame ([gvGetCodecInfo](#) can be used to get the size of an encoded frame). The function will fill this memory with as many encoded frames as it can. The [len](#) parameter must point to an int which is set to the maximum number of bytes that can be written to the block of memory pointed to be the packet parameter. After the function returns, if it was successful, [len](#) will point to the number of bytes that were written to the block of memory. Also, the [frameStamp](#) parameter will point to the frame stamp for the captured packet and the volume parameter will point to the peak volume for the audio in the frame. The volume ranges from 0.0 to 1.0, and it can be used to power a per-player graphic voice activity meter. If the function succeeds in getting a packet and encoding it into the provided memory block, it will return [GVTrue](#). If it returns [GVFalse](#), then there was either no audio data available to capture or some sort of error capturing the audio.

Once a packet has been captured, it is ready to be sent to other players. It is the application's responsibility to route the packet to its final destinations. It should also route the packet's frame stamp along with the packet. The application can choose to not send the packet anywhere, in effect muting the player. It can pass it back to the SDK to be played locally. It can send the packet directly to one or more other players for them to play. It can send the packet to a server which would then decide, possibly based on information such as players' positions or teams, who the packet should be sent to.

You can check how many bytes are available for capture before calling `gvCapturePacket`, using `gvGetAvailableCaptureBytes`.

```
int gvGetAvailableCaptureBytes(GVDevice device);
```

Simply provide the handle to a capture device and it will return the number of bytes that are currently available for capture. To determine the number of encoded frames that this is, divide the return value by the number of bytes in an encoded frame (which you can get with `gvGetCodecInfo`). Note that even if there are bytes available, `gvCapturePacket` may return `GVFalse`. This could happen if a capture threshold has been set, and the voice audio does not cross the threshold. In that case GV would skip over that captured audio, and its bytes would no longer count as available bytes.

Playing Packets

When the application receives a packet that it wants to play, it should pass it to `gvPlayPacket`.

```
void gvPlayPacket(GVDevice device, const GVByte * pa
```

The first parameter tells GV which device to use for playing the packet. The second parameter is pointer to the packet. And the next three parameters provide GV with the packet's length, the source that originally spoke the audio, and the packet's frame stamp. GV will schedule the packet to be played soon. A short delay is added which enables the packets to be synchronized before they are played, allowing for variations

in Internet transit time and application timing. The packet is synchronized based on its source, so you must ensure that each unique talker has his own unique source, and all packets are played using the correct source. Note that the same packet can be played on multiple playback devices.

`gvPlayPacket` only schedules a packet to be played in the future. The application must also call `gvThink` on a regular basis to ensure that the packets are actually played.

```
void gvThink(void);
```

`gvThink` will check, for each device, how much space has become available for writing in the playback buffer (which may or may not be on the actual sound hardware). It will then check to see if the device has any sources that have audio which should be played during the time period that the newly available space represents. If so, the audio will be mixed into the playback buffer, and the audio will then be played when the playback position reaches that point in the buffer. If the playback device is stopped before that happens, then the audio will not be played, even if the device is then restarted.

`gvThink` should generally be called once for each run through the application's main loop, or approximately every 10-30ms. If it is not called often enough, the playback position will reach a point in the playback buffer that GV has not yet had a chance to mix to, resulting in an audible skipping effect.

Local Echo

To get local echo with GV, simply pass captured packets to a playback device. You'll need to ensure that you pass `gvPlayPacket` a `GVSOURCE` that will not conflict with any of the remote talkers' sources. There will be a slight delay due to the fact that the audio is being processed by both GV and the application. Local echo will generally not be desired for regular use, but is very useful when a user is selecting and configuring devices.

Talking Sources

To determine if a particular source is currently talking, call [gvIsSourceTalking](#).

```
GVBool gvIsSourceTalking(GVDevice device, GVSource s
```

It will return [GVTrue](#) if the source is talking on the specified device. To get a list of all of the sources that are currently talking on a particular device, use [gvListTalkingSources](#).

```
int gvListTalkingSources(GVDevice device, GVSource s
```

The [maxSources](#) parameter should be the number of sources that can be stored in the sources array. The function will return the number of sources that were talking on the device, and it will store their [GVSources](#) in the sources array. 0 will be returned if there are no sources talking.

GV has a hardcoded limit that does not allow more than 8 sources to talk simultaneously. This allows it to preallocate memory that it needs to store for a source while it is talking. A user will typically not understand more than 2 or 3 users talking simultaneously, so the limit should be high enough. If the application attempts to play audio from more than 8 sources at a time, audio for the 9th source will be automatically dropped.

Muting Sources

All you need to do to mute a source in GV is ignore packets that originate with that source. Optionally you can send a message to the source telling him to stop sending you packets, which will cut down on network bandwidth. But the only requirement for muting is that the packets are not played.

Threshold

After a capture device has been started, it will continually generate packets and pass them to the application through [gvCapturePacket](#). If a capture threshold is set on a capture device, then a packet will only be passed to the application if its peak volume is at least as high as the capture threshold. Use [gvSetCaptureThreshold](#) to set the threshold.

```
void gvSetCaptureThreshold(GVDevice device, GVScalar
```

The range for `threshold` is 0.0 to 1.0. A value of approximately 0.10 to 0.15 will generally work well, although ideally the user should have a way to configure the threshold. GV will continue passing packets to the application for about half a second after the peak volume drops below the threshold. This helps to catch speech in which the level trails off or has a small dip. The default `threshold` is 0.0, which means that all audio will be considered over the threshold and will be captured. To remove a threshold that has been set, simply call this function again with a `threshold` of 0.0.

Use `gvGetCaptureThreshold` to get the current value of `threshold`.

```
GVScalar gvGetCaptureThreshold(GVDevice device);
```

Volume

Use `gvSetDeviceVolume` to apply a volume control to a capture or playback device.

```
void gvSetDeviceVolume(GVDevice device, GVDeviceType
```

The `volume` range is 0.0 to 1.0. The `type` parameter controls if this gets set as a capture volume (`GV_CAPTURE`), a playback volume (`GV_PLAYBACK`), or for both capture and playback (`GV_CAPTURE_AND_PLAYBACK`). To get the volume use `gvGetDeviceVolume`.

```
GVScalar gvGetDeviceVolume(GVDevice device, GVDevice
```

It will return the `volume` in a range from 0.0 to 1.0. For a device that supports both capture and playback, this function can only be used to get either the capture volume or the playback volume, not both.

Advanced Implementation

This section describes some of the more advanced functionality of GV.

Detecting Devices Being Unplugged

A [gvUnpluggedCallback](#) allows an application to know if a device is unplugged or otherwise stops working.

```
typedef void (* gvUnpluggedCallback)(GVDevice device
```

The callback function is passed the handle to the device that was unplugged. The device will be freed by GV immediately after this function returns.

Use [gvSetUnpluggedCallback](#) to set the unplugged callback for a particular device.

```
void gvSetUnpluggedCallback(GVDevice device, gvUnplu
```

Custom Device

A custom device allows an application to supply its own audio hardware interface. To create a custom device, use [gvNewCustomDevice](#).

```
GVDevice gvNewCustomDevice(GVDeviceType type);
```

Specify if the custom device will handle capture ([GV_CAPTURE](#)), playback ([GV_PLAYBACK](#)), or both capture and playback ([GV_CAPTURE_AND_PLAYBACK](#)). The function will return a [GVDevice](#) if it is successful, or [NULL](#) if it cannot create the device. When an application has finished using a custom device, it should call [gvFreeDevice](#) to free its resources.

All custom devices must interface with GV using the defined sample rate ([GV_SAMPLES_PER_SECOND](#)) and bit rate ([GV_BITS_PER_SAMPLE](#)). The default sample rate is 8000Hz, and the default bits per sample is 16.

Custom devices will process frames of audio as quickly as you want. In other words, the application controls a custom device's clock rate. Because of this, you have to be sure that you request or provide samples at the correct sample rate. In other words, you should be going through approximately 8000 samples per second. Timing variations in the short run are not a problem (for example, if the system's clock is only accurate to within 30ms), but over time the sample rate must average out to approximately the correct rate (in a minute you should have requested approximately $60 * 8000 = 480000$ samples). GV compensates for drift between clocks (known as clock skew), however with a custom device the application is still responsible for synching with the system clock.

If you are not using GV's default audio device support, you can define [GV_NO_DEFAULT_HARDWARE](#). This will remove all internal references to audio hardware. That way, you don't need to setup DirectX or lgAud if you won't be using it. If [GV_NO_DEFAULT_HARDWARE](#) is defined, then [gvListDevices](#) and [gvNewDevice](#) will have no purpose, so they will not be included and cannot be called.

Custom Playback

With a custom playback device, you are taking over the function of a hardware sound device. GV delivers a mixed audio stream, which you can do whatever you want with, such as playing through a sound device, saving to disk, or displaying visually.

You provide a custom playback with packets the same way you would any other playback device, by using [gvPlayPacket](#). You can also use all of the other standard playback device functions, such as [gvIsSourceTalking](#), and [gvSetDeviceVolume](#). However you do need not call [gvThink](#) with custom devices. Instead you use [gvGetCustomPlaybackAudio](#).

```
GVBool gvGetCustomPlaybackAudio(GVDevice device, GVS
```

The `numSamples` parameter specifies how many samples can be written to the memory pointed to by `audio`. `numSamples` must be a multiple of the samples per frame for the current codec (which you can check using `gvGetCodecInfo`). This is because GV mixes audio a frame at a time. The function will return `GVTrue` if it was able to fill the memory with audio.

Custom Capture

A custom capture device allows you to provide your own sound source in GV. It does this by replacing `gvCapturePacket` with a function that both provides GV with captured audio and then gets a packet containing that audio. This function is `gvSetCustomCaptureAudio`.

```
GVBool gvSetCustomCaptureAudio(GVDevice device, cons
```

The `audio` parameter points to the incoming audio stream, and `numSamples` is the number of samples to capture. `numSamples` must be a multiple of the samples per frame for the current codec (which you can check using `gvGetCodecInfo`). The audio will be encoded into a packet, which will be stored at the memory pointed to by the `packet` parameter. `packetLen` must be large enough to hold all of the encoded frames supplied by the `audio` parameter. If the function successfully encodes the audio into the packet, it returns `GVTrue`. It will return `GVFalse` if a threshold is set and the audio's peak volume did not cross the threshold. If the function succeeds, `packetLen` will store the number of bytes encoded into packet, `frameStamp` will point to the frame stamp for the packet, and `volume` will point to the peak volume for the packet.

Note that although the `audio` parameter is a `const`, if a capture volume or a capture filter is set on this device, then the memory will be modified. This is done to minimize the amount of memory and copying needed.

Custom capture devices can be used the same way as other capture

devices. For example, you can use a capture threshold or set a volume. However there are two functions that are not supported by custom capture devices: [gvGetAvailableCaptureBytes](#) and [gvCapturePacket](#). That is because you are now providing the capture interface.

Custom Codec

An application can use a codec other than the ones provided with GV, by using a custom codec. To start using a custom codec, first fill in a [GVCustomCodecInfo](#) structure.

```
typedef struct
{
    int m_samplesPerFrame;
    int m_encodedFrameSize;

    GVBool (* m_newDecoderCallback)(GVDecoderData * data
    void (* m_freeDecoderCallback)(GVDecoderData data);
    void (* m_encodeCallback)(GVByte * out, const GVSamp
    // decode must _add_ to, not set the output
    void (* m_decodeCallback)(GVSample * out, const GVBy
} GVCustomCodecInfo;
```

[m_samplesPerFrame](#) is the number of samples that this codec expects in a raw (unencoded) frame of audio. This can be whatever value is used by the codec, however it is typically about 160 samples.

[m_encodedFrameSize](#) is the number of bytes in an encoded frame of audio produced by this codec. The ratio of the samples per frame and encoded frame size is directly related to the codec's output stream bit rate.

[m_newDecoderCallback](#) is used to allocate a new decoder instance for each incoming source. Some codecs do not require this, and they should set the [m_newDecoderCallback](#) member to `NULL`. For codecs that do require per-source data, they should allocate a new decoder data state and set the data parameter to point to it, then return `TRUE`. If they cannot allocate a new decoder data, then they should return `GVFalse`.

The `m_freeDecoderCallback` is used to free decoder data allocated through `m_newDecoderCallback`. If a codec set `m_newDecoderCallback` to `NULL`, it should set `m_freeDecoderCallback` to `NULL` as well. Otherwise it should provide a function that frees the decoder data.

`m_encodeCallback` is used to encode data. The `in` parameter points to the input data, with is a single raw frame of samples. The number of samples passed to this function will always be `m_samplesPerFrame`. The `out` parameter points to the memory into which the callback should encode the input data. The memory will always be large enough to hold one frame of compressed audio, which will always be `m_encodedFrameSize` bytes long.

`m_decodeCallback` is used to decode data. The `in` parameter will point to an encoded frame of audio, which will be `m_encodedFrameSize` bytes long. The `out` parameter which will be large enough to hold `m_samplesPerFrame` samples of audio. The decoder data is provided for codecs that need it. The important thing to know with the decode callback is that it should not decode directly into the `out` buffer, but it should add to it. This allows GV to decode and mix at the same time, without having to decode into a temporary buffer which would then be mixed into the output stream.

Once you have filled in a `GVCustomCodecInfo` structure, use `gvSetCustomCodec` to set it as the codec. It will replace any codec that has been set with `gvSetCodec`.

```
void gvSetCustomCodec(GVCustomCodecInfo * info);
```

If you are not using GV's default codec support, you can define `GV_NO_DEFAULT_CODEC`. This will remove all internal references to the default codec. That way, you don't need to setup Speex or lgCodec if you won't be using it. If `GV_NO_DEFAULT_CODEC` is defined, then `gvSetCodec` will have no purpose, so it will not be included and cannot be called.

Filter

Filtering allows you to process, or just monitor, audio that has been captured or is being played. A filter callback, prototyped as the [gvFilterCallback](#) type, is passed the device the filtering is happening on, the audio to filter, and the audio's frame stamp. The audio will always be a single raw frame of audio. Use [gvGetCodecInfo](#) to get the number of samples in a raw frame.

```
typedef void (* gvFilterCallback)(GVDevice device, G
```

The callback can modify the audio in any way that it wants. However once the function returns it can no longer access the audio. For capture devices, audio will only be passed to the filter if it crosses the threshold (if one is set). For playback devices, audio is filtered after all of the sources have been mixed.

To set a filter on a device, use [gvSetFilter](#).

```
void gvSetFilter(GVDevice device, GVDeviceType type,
```

You can use the function to set a filter on any device, and to set it for capture or playback. A device can have only one capture filter at a time and only one playback filter at a time. To clear a filter, call this function with the callback set to [NULL](#).

Device Wizard

With Win32 only, a device setup wizard is available if the user has DirectX 8 or greater. It can be instantiated for a pair of devices using [gvRunSetupWizard](#).

```
GVBool gvRunSetupWizard(GVDeviceID captureDeviceID,
```

The function takes a capture device ID and a playback device ID. This function cannot be called successfully if these devices have been initialized with [gvNewDevice](#) - [gvRunSetupWizard](#) must be called first. The wizard will take over control of the program while it executes. It has the user speak into the capture device, and monitors the audio to

automatically set system level capture and playback volumes. It will return `GVTrue` if the user successfully completes the wizard, `GVFalse` otherwise. If the wizard is successful, DirectX will store the results in the registry. Once it does this, `gvAreDevicesSetup` can be used to determine if the registry has information on the two specified devices. If so, it returns `GVTrue`, and the wizard does not need to be run again. If it returns `GVFalse`, then the wizard has not been run for the pair of devices.

```
GVBool gvAreDevicesSetup(GVDeviceID captureDevice, G
```

Playback Delays

There are two delays that occur between when the application gives GV audio to play and the audio is actually heard. The first delay is a synchronization delay added automatically by GV. The second is a result of missing the audio into the hardware before it needs to play it, which ensures that the hardware always has audio ready to play, resulting in no audible gaps.

Synchronization Delay

The synchronization delay is controlled by the `GVI_SYNCHRONIZATION_DELAY` define at the top of `gvSource.c`. This controls how many milliseconds GV will wait before mixing the audio into the playback buffer. This allows for variations in Internet transit timing and in application timing. For example, if one packet takes 100 ms to arrive and the next takes 150 ms, the delay will ensure that the packets can still be played back as one smooth stream.

The greater the value of the delay, the longer the lag will be between when something is spoken and when it is heard. If the lag gets to be too large, it can be very noticeable to users. However a greater delay also allows for more variation in timing, resulting in a smoother experience on systems with a large amount of timing variation.

Playback Buffer Size

The playback buffer size controls approximately how far in advance the audio will be mixed into the playback buffer. This is because, as the playback position moves through the playback buffer, the space immediately behind the playback position becomes available for writing audio data that will be played when the position loops all the way back again. GV writes into the memory behind the playback position as soon as it becomes available. This ensures that there will be no skipping effect, which results from the playback position getting to a point that has not yet had new audio written to it. The user will hear the audio that was written to the buffer for the last loop, resulting in an audible glitch.

The buffer size is specified by the `GVI_PLAYBACK_BUFFER_MILLISECONDS` define, which is in `gvDevice.h`. If a custom playback device is being used, then the application is doing any audio buffering, so the define has no effect.

The most important thing that affects buffer size is the amount of time between calls to `gvThink`, which is when audio gets mixed into the playback buffer. The larger the buffer size, the longer the delay between when something is said and when it is heard. However a larger size allows for longer delay between calls to `gvThink`. With a smaller buffer size `gvThink` must be called more often, to ensure that the playback position does not loop all the way around without GV having a chance to write new audio data to it. In general, the buffer size should be at least twice as long as the maximum time between calls to `gvThink`.

Global Focus

On Win32 only, the default is for the playback device to have global focus. This means that even if the user switches to a different application, audio played through GV will still be heard. However if you define `GV_NO_GLOBAL_FOCUS`, then playback won't be heard if the application's window loses focus.

Excluding PS2 Hardware Types

You can exclude support for particular hardware device types on the PS2 by compiling the SDK with one or more defines set. Use `GV_NO_PS2_SPU2` to exclude SPU2 support, `GV_NO_PS2_HEADSET` to

exclude IgAud supprt (USB audio devices - headset/microphone/speakers), and/or define `GV_NO_PS2_EYETOY` to exclude IgVid support (EyetoY).

Performance Data

Bandwidth and CPU Usage for Codecs

OS	Codec	Bandwidth (bits-per-second)	CPU Usage (encoding %)*	CPU Usage (decoding %)*
Win32	Super Low Bandwidth	3600	2.4	0.2
	Low Bandwidth	5600	1.4	0.2
	Average	8000	1.8	0.2
	High Bandwidth	14800	2.1	0.2
	Super High Bandwidth	24400	3.0	0.2
Any	No Compression	128000	0.0	0.0
PS2	Super Low Bandwidth	2489	4.5	18.8
	Low Bandwidth	8000	X**	X**
	Average	13200	2.8	1.5
	High Bandwidth	24000	4.1	3.8
	Super High Bandwidth	64000	0.1	0.1
PS3	Super Low Bandwidth	3600	3.6	0.4
	Low Bandwidth	5600	2.7	0.4
	Average	8000	3.5	0.4
	High Bandwidth	14800	4.5	0.4
	Super High Bandwidth	24400	6.0	0.4

*CPU Usage for Win32 was tested on a Pentium 4 2.8Ghz PC with 1GB

of ram and WinXP SP1.

**This codec is currently not working correctly on the PS2.

PS2 Memory Usage

Note that this memory usage information only applies to the PS2.

The SDK uses about 40-50KB of memory on the EE, and about 25-50KB of IOP memory. The exact amount of memory depends on which codec is used, which hardware types are supported (by default, all are supported), and which devices are used at runtime. All memory allocations take place in calls to [gvStartup](#), [gvSetCodec](#), and [gvNewDevice](#). The SDK does not allocate additional memory while it is thinking, capturing, or playing packets. All allocated memory is freed in calls to [gvFreeDevice](#) and [gvShutdown](#).

gvStartup

Under 1KB of EE memory is allocated at startup. This is used by IgAud and IgVid.

gvSetCodec

When a codec is set, both IgCodec and GV allocate EE memory. The exact amount depends on the codec. The Super Low Bandwidth codec uses about 20KB, the Average codec uses about 15KB, the High Quality codec uses about 16KB, and the Super High Quality codec uses about 35KB.

gvNewDevice

When a new device is instantiated with a call to [gvNewDevice](#), both GV and IgAud or IgVid, if they are used, allocate EE memory. Memory is also allocated on the IOP, where the actual interaction with the device takes place. In addition SPU2 support uses about 25KB of static memory on the EE. The exact amount of allocated memory depends on the hardware type. A SPU2 device uses less than 1KB of EE memory and about 26KB

of IOP memory. A USB audio device (headset/microphone/speakers) uses about 1KB of EE memory. It will also use about 4KB of IOP memory if it is used for playback, about 24KB of IOP memory if it is used for capture, and about 28KB of IOP memory if it is used for both capture and playback.

If you will not be supporting one or more types of PS2 hardware devices, you can save the memory that it would use by excluding it from the SDK. See the Advanced section for more information.

PS3 Memory Usage

The Voice SDK was tested on the PS3 PPU using a USB Headset.

The PS3 memory consumption when using no compression showed to be around 70KB. Using the Speex codec with settings Super High quality, High quality, Average, Low bandwidth, Super Low bandwidth used about 16KB, 12KB, 8KB, 7KB, 6KB respectively.

Appendix A: Voice SDK with Speex Codec on PS3 SPUs (***BETA***)

Background/Overview

This appendix assumes that all readers have experience with cell programming, and the SPURS library. Another assumption is that the development environment. The Voice SDK already has support for encoding and decoding using the Speex codec on the PPU processor (considered the main processor). It now supports encoding and decoding operations using Speex on the SPU processors. The SPURS library was the best choice, and we selected the SPURS taskset model. The SPURS taskset model involves coarse-grain processing on SPUs, which fits with Speex. The purpose of using SPUs via SPURS tasksets to perform encoding/decoding is to reclaim PPU processing time, allowing developers to use it for other purposes such as game logic, graphics code, etc. To get the most benefit from this Voice SDK feature, developers **should have the Voice SDK calls that require the PPU in a separate PPU thread--The reason for this is because the Voice SDK synchronously performs encoding and decoding.** Thus, putting the Voice SDK in a separate PPU thread will allow the game operations mentioned previously to execute without having to wait for encoding/decoding to complete on the Voice SDK thread. We also recommend developers put all Gamespy calls in a separate thread, which can be the same thread as the Voice SDK thread. Another important thing about DMA bandwidth, to take is that the SDK is not anywhere near the maximum DMA transfer speed. This is because of the low memory requirements displayed next.

Memory Consumption: Maximum Bytes consumed

Codec Quality	8 Khz Audio	16 Khz Audio
SuperHighQuality	123448	133848
HighQuality	118648	124248
Average	114648	118648
LowBandwith	113848	116248

SuperLowBandwith	113048	115448
------------------	--------	--------

Memory consumption was measured using the Voice2Test application for the PS3 (Voice2\Voice2Test\gvps3prodgspeexspu). The application was setup to use local echo to ensure both encoding and decoding occurred. The Voice SDK uses varying amounts of memory because it has to allocate buffers to keep track of the encoder/decoder state and to perform DMA transfers easily. Most of the memory consumed in these cases were allocations for the encoder/decoder state. Since most games will only allow one person to talk at a time, memory usage should be close to these figures.

PPU Impact for one frame (20 ms)

Codec Quality	8 KHz (Encode)	8 KHz (Decode)	16 KHz (Encode)	16 KHz (Decode)
SuperHighQuality	0.37%	0.34%	0.47%	0.45%
HighQuality	0.36%	0.34%	0.40%	0.42%
Average	0.36%	0.34%	0.37%	0.37%
LowBandwith	0.31%	0.31%	0.35%	0.35%
SuperLowBandwith	0.35%	0.31%	0.36%	0.35%

SPU Impact for one frame (20 ms)

Codec Quality	8 KHz (Encode)	8 KHz (Decode)	16 KHz (Encode)	16 KHz (Decode)
SuperHighQuality	10.5%	0.75%	23.86%	1.52%
HighQuality	7.57%	0.84%	18.00%	1.53%
Average	7.1%	0.82%	12.09%	1.4%
LowBandwith	5.78%	0.89%	6.86%	1.36%
SuperLowBandwith	6.02%	0.89%	11.46%	1.34%

The percentages listed above show the average portion of each frame spent on each processor. Sony's PA Suite and Voice2Test for the PS3 were used to calculate these percentages. According to the PPU/SPU impact charts, the Voice SDK does not utilize the PPU as much as the

SPU. The PPU times reflect the amount of time spent in the encode/decode call that occurs internally in the Voice SDK which does not have processor intensive code. The rest of the time was spent on: the SPU performing the encode/decode operations, and the PPU mostly idling. Again, it is recommended that the Voice SDK is used in a PPU thread separate from the main thread because the PPU has to block until the SPU completes encode/decode operations and idles most of the time.

DMA Bandwidth (measured in bytes)

DMA Data to and from SPU (one trip)	8 KHz (Encode)	8 KHz (Decode)	16 KHz (Encode)	16 KHz (Decode)
Encoded Buffer	128	128	128	128
State Buffer	64672	41024	64672	41024
Decoded Buffer	160	160	320	320
Task Descriptor	46	46	46	46
Task Output	20	20	20	20

Bytes per second used (50 frames/s * 2 transfers)	6502600	4137800	6518600	4153800

The DMA bandwidth metrics above account for all DMA transfers that occur during encode/decode operations. These metrics are based on a single frame, and are considered a one way trip. The state buffers remain the same for both encoding and decoding even when using different sampling rates. However, the buffer to perform DMA transfer of decoded audio vary depending on the sample rate. The bytes per second shows the bandwidth that is used for a full second of audio. It consists of 50 frames multiplied by two transfers. These transfers account for a round trip. The DMA transfer performance according to Sony's documentation is 13.2 GB/s for reads, and 22.8 GB/s for writes.

The numbers in the chart above are nowhere near the performance of DMA transfers that Sony mentions in their documentation. Therefore, there should be plenty of room for other SPURS tasks to occur in addition

to this.

Implementation

Speex Task and Speex Task Manager

The Speex Task performs encode and decode operations using the speex codec, while the Speex Task Manager facilitates communication between the Voice SDK and the Speex Task. Both of these projects are required. They are available by downloading the GameSpy Common Code. The Speex Task Manager and the Speex Task are located in the following folders:

```
Gamespy\common\ps3\SpeexSpursTaskManager  
Gamespy\common\ps3\SpeexSpursTaskManager\SpeexSpursT
```

The Voice SDK depends on these two projects and a **new speex interface: gvSpeexSpu.c**. These projects should be added to your Visual Studio solution. After adding the two projects to your solution, set the project dependencies to include these two projects for the project that has the Voice SDK. Refer to the Voice2Test sample that is setup for Speex SPU usage: [Voice2\Voice2Test\gvps3prodgspeexspu](#).



As previously mentioned, extract the Speex codec into the Voice SDK folder in order to avoid seeing errors and warnings about missing Speex codec files. The rest of Voice SDK implementation can be found above.

Passing in an Existing Spurs Instance

In Voice2Test.c, there is an example of **an existing SPURS instance** being passed to a helper function. The Speex Spurs Task Manager includes the function. The game should have an initialized spurs instance, and pass it to the function [spursConfiguration_initWithSpurs](#). Make sure to include the file "**spursConfiguration.h**" prior to calling it. The function also takes in the number of SPUs to use for the Speex task, and the priorities of the

task per SPU. This can be useful for developers that want to control the amount SPUs to dedicate to a given SPURS task. Here is an excerpt of the Voice2Test.c that shows usage of the function `spursConfiguration_initWithSpurs`:

```
#if defined(USER_CREATED_SPURS_INSTANCE)
#include "spursConfiguration.h"
#include <sys/spu_initialize.h>
int iReturn, iNumSpus, ppuThreadPriority, spuThreadP
bool exitIfNoWork;
CellSpurs* myCellSpurs;
uint8_t auLocalPriorities[8]={1,1,1,1,1,1,1,1};
#endif

...

static GVBool Initialize(const char * remoteIP)
{
...
    iNumSpus = 1;
    spuThreadPriority = 200;
    ppuThreadPriority = 1000;
    exitIfNoWork = false;
    myCellSpurs = (CellSpurs*) gsimemalign(128,

    // initializing spus themselves before using
    sys_spu_initialize(iNumSpus,0);

    iReturn=cellSpursInitialize(myCellSpurs, iNu
    if (iReturn!=CELL_OK)
    {
        printf("Error initializing spus\n")
        return GVFalse;
    }
    spursConfiguration_initWithSpurs(myCellSpurs

...
}
```

Embedding the Speex Task with your project

To Embed the Speex Task, There is an easy way to accomplish this. Simply make the SpeexSpursTask project a dependency of the project containing the Voice SDK. Developers that like to go further can refer to the Post-Build configuration options in the SpeexSpursTask project. There is also a section in the SPURS tutorial for developers that wish to go further in customizing the embedding. Please refer to the SPURS Tutorial 3.5: Building the Program in Sony's Documentation.

Voice SDK Functions

gvAreDevicesSetup	Determines if the registry has information on the specified device pair.
gvCapturePacket	Takes captured audio data out of the internal capture buffer, storing it in the provided packet memory block.
gvCleanup	Performs any necessary internal cleanup. GV cannot be used again until gvStartup is called.
gvFreeDevice	Frees a device so that GV can clean it up
gvGetAvailableCaptureBytes	Discovers how many bytes are currently available for capture on the given device.
gvGetCaptureMode	Gets the capture mode for the device.
gvGetCaptureThreshold	Gets the current value of the capture threshold for the device
gvGetCodecInfo	Obtains the particular stats for the codec.
gvGetCustomPlaybackAudio	Retrieves any audio data that is ready to be played through a custom playback device.
gvGetDeviceVolume	Gets the volume from a capture or playback

	device.
gvGetGlobalMute	Gets the current status of global mute.
gvGetPushToTalk	Tells you if PushToTalk is currently turned on or off.
gvIsDeviceStarted	Checks to see if a whether or not a device has been started as the given device type.
gvIsSourceTalking	Determines if a particular source is currently talking on the specified device.
gvListDevices	Gets a list of devices available on the system
gvListTalkingSources	Gets a list of all of the sources that are currently talking on a particular device.
gvNewCustomDevice	Creates a custom device, which allows an application to supply its own audio hardware interface.
gvNewDevice	Initializes a device
gvPlayPacket	Plays a packet retrieved from the capture buffer.
gvRunSetupWizard	Interacts with the user to set up the capture and playback devices.

gvSetCaptureMode	Sets the capture mode for the device.
gvSetCaptureThreshold	Sets the threshold volume on a device. A packet will only be passed to the application if its peak volume is at least as high as the capture threshold.
gvSetCodec	Sets the codec to be used by the SDK.
gvSetCustomCaptureAudio	For a custom capture device, encodes captured audio from a stream into a packet, storing it at provided memory.
gvSetCustomCodec	Tells GV to use an application-provided codec instead of a built-in codec.
gvSetDeviceVolume	Sets a device's volume.
gvSetFilter	Sets a device's filter callback.
gvSetGlobalMute	Sets the global mute value - defaults to false.
gvSetPushToTalk	Used to turn on or off capturing for a device. Must be in GVCaptureModePushToTalk mode.
gvSetUnpluggedCallback	Sets a callback to be called when the SDK detects that the device was unplugged or is

	no longer functioning.
gvStartDevice	Starts a device capturing and/or playing audio.
gvStartup	Initializes the SDK.
gvStopDevice	Stops a device that is capturing and/or playing audio.
gvThink	Allows playback devices to play audio scheduled for playback.

gvAreDevicesSetup

Determines if the registry has information on the specified device pair.

```
GVBool gvAreDevicesSetup(  
    GVDeviceID captureDeviceID,  
    GVDeviceID playbackDeviceID );
```

Routine	Required Header	Distribution
gvAreDevicesSetup	<gv.h>	SDKZIP

Return Value

Returns GVTrue if gvRunSetupWizard has been run on the pair; otherwise, GVFalse.

Parameters

captureDeviceID

[in] Reference to the device used to capture audio

playbackDeviceID

[in] Reference to the device used to playback audio

Remarks

If **gvAreDevicesSetup** returns GVTrue, and the gvRunSetupWizard does not need to be run again. If it returns GVFalse, then the gvRunSetupWizard has not been run for the pair of devices.

Section Reference: [Gamespy Voice SDK](#)

See Also: [gvNewDevice](#), [gvRunSetupWizard](#)

gvCapturePacket

Takes captured audio data out of the internal capture buffer, storing it in the provided packet memory block.

```
GVBool gvCapturePacket(  
    GVDevice device,  
    GVByte * packet,  
    int * len,  
    GVFrameStamp * frameStamp,  
    GVScalar * volume );
```

Routine	Required Header	Distribution
gvCapturePacket	<gv.h>	SDKZIP

Return Value

GVTrue if successful in getting a packet and encoding it into the provided memory block; GVFalse if either no audio data was available to capture or an error occurred while capturing the audio.

Parameters

device

[in] A handle to the capture device

packet

[in] A block of memory to receive the data

len

[ref] The maximum / number of bytes moved into the memory block specified by packet parameter.

frameStamp

[out] The frame stamp for the captured packet

volume

[out] The peak volume for the audio in the frame

Remarks

The packet parameter must be large enough to hold at least one encoded frame (`gvGetCodecInfo` can be used to get the size of an encoded frame). The function will fill this memory with as many encoded frames as it can.

The `len` parameter must point to an `int` which is set to the maximum number of bytes that can be written to the block of memory pointed to be the packet parameter. After the function returns, if it was successful, `len` will point to the number of bytes that were written to the block of memory. The `frameStamp` parameter will receive the frame stamp for the captured packet, and the `volume` parameter, the peak volume for the audio in the frame. The volume ranges from 0.0 to 1.0, and it can be used to power a per-player graphic voice activity meter.

Section Reference: [Gamespy Voice SDK](#)

See Also: [gvGetAvailableCaptureBytes](#), [gvGetCodecInfo](#)

gvCleanup

Performs any necessary internal cleanup. GV cannot be used again until gvStartup is called.

void gvCleanup();

Routine	Required Header	Distribution
gvCleanup	<gv.h>	SDKZIP

Section Reference: [Gamespy Voice SDK](#)

See Also: [gvStartup](#)

gvFreeDevice

Frees a device so that GV can clean it up.

```
void gvFreeDevice(  
    GVDevice device );
```

Routine	Required Header	Distribution
gvFreeDevice	<gv.h>	SDKZIP

Parameters

device

[in] The handle to the device to be freed.

Remarks

Once a device has been freed it can no longer be used. After calling this function you should set the variable in which you stored the device handle to NULL.

Section Reference: [Gamespy Voice SDK](#)

See Also: [gvNewDevice](#), [gvStartDevice](#), [gvStopDevice](#)

gvGetAvailableCaptureBytes

Discovers how many bytes are currently available for capture on the given device.

```
int gvGetAvailableCaptureBytes(  
    GVDevice device );
```

Routine	Required Header	Distribution
gvGetAvailableCaptureBytes	<gv.h>	SDKZIP

Return Value

Returns the number of bytes available.

Parameters

device

[in] The handle to the device.

Remarks

To determine the number of encoded frames, divide the return value by the number of bytes in an encoded frame (which you can get with `gvGetCodecInfo`).

Note that even if there are bytes available, `gvCapturePacket` may return `GVFalse`. This could happen if a capture threshold has been set, and the voice audio does not cross the threshold. In that case GV would skip over that captured audio, and its bytes would no longer count as available bytes.

Section Reference: [Gamespy Voice SDK](#)

See Also: [gvCapturePacket](#), [gvGetCodecInfo](#)

gvGetCaptureMode

Gets the capture mode for the device.

```
GVCaptureMode gvGetCaptureMode(  
GVDevice device );
```

Routine	Required Header	Distribution
gvGetCaptureMode	<gv.h>	SDKZIP

Return Value

The current capture mode for this device.

Parameters

device

[in] The handle to the capture device.

Section Reference: [Gamespy Voice SDK](#)

See Also: [gvSetCaptureMode](#), [GVCaptureMode](#)

gvGetCaptureThreshold

Gets the current value of the capture threshold for the device.

```
GVScalar gvGetCaptureThreshold(  
    GVDevice device );
```

Routine	Required Header	Distribution
gvGetCaptureThreshold	<gv.h>	SDKZIP

Return Value

Returns the current threshold value.

Parameters

device

[in] The handle to the capture device.

Remarks

Retrieves the value that was assigned by `gvSetCaptureThreshold`.

Section Reference: [Gamespy Voice SDK](#)

See Also: [gvSetCaptureThreshold](#)

gvGetCodecInfo

Obtains the particular stats for the codec.

```
void gvGetCodecInfo(  
    int * samplesPerFrame,  
    int * encodedFrameSize,  
    int * bitsPerSecond );
```

Routine	Required Header	Distribution
gvGetCodecInfo	<gv.h>	SDKZIP

Parameters

samplesPerFrame

[out] The samples per frame.

encodedFrameSize

[out] The encoded frame size in bytes

bitsPerSecond

[out] The bits per second.

Remarks

Note that the bits per second doesn't take into account any overhead, such as the need to transmit a frame stamp value with each packet. It is based only on the encoded frame size and the number of frames per second.

Section Reference: [Gamespy Voice SDK](#)

See Also: [gvSetCodec](#)

gvGetCustomPlaybackAudio

Retrieves any audio data that is ready to be played through a custom playback device.

```
GVBool gvGetCustomPlaybackAudio(  
    GVDevice device,  
    GVSample * audio,  
    int numSamples );
```

Routine	Required Header	Distribution
gvGetCustomPlaybackAudio	<gv.h>	SDKZIP

Return Value

GVTrue if the audio buffer was filled, GVFalse otherwise.

Parameters

device

[in] The custom playback device from which to retrieve audio.

audio

[out] Buffer to fill with audio samples to be played by the custom device.

numSamples

[in] Size of the audio buffer in samples. Must be a multiple of the current `samplesPerFrame`.

Remarks

numSamples must be a multiple of the samples per frame for the current codec (which you can check using gvGetCodecInfo). This is because GV mixes audio a frame at a time.

This function should be called at the same rate at which the custom playback device is actually playing audio. In other words, the physical device should be pulling data with this function when it needs it - the data is not being pushed to the physical device. This is because the SDK compensates for differences in audio clock rates, and calling it at the correct rate will ensure a consistent rate of playback.

The GV_SAMPLES_PER_SECOND and GV_BITE_PER_SAMPLE defines can be used to determine the sample rate and bitrate of the audio.

Section Reference: [Gamespy Voice SDK](#)

See Also: [gvNewCustomDevice](#)

gvGetDeviceVolume

Gets the volume from a capture or playback device.

```
GVScalar gvGetDeviceVolume(  
    GVDevice device,  
    GVDeviceType type );
```

Routine	Required Header	Distribution
gvGetDeviceVolume	<gv.h>	SDKZIP

Return Value

Returns the specified volume for the device.

Parameters

device

[in] The handle to the device.

type

[in] Specifies either the playback volume or the capture volume

Remarks

The volume range is 0.0 to 1.0.

The type parameter controls if this gets set as a capture volume (GV_CAPTURE), a playback volume (GV_PLAYBACK). For a device that supports both capture and playback, this function can only be used to get either the capture volume or the playback volume, not both.

Section Reference: [Gamespy Voice SDK](#)

See Also: [gvSetDeviceVolume](#)

gvGetGlobalMute

Gets the current status of global mute.

GVBool gvGetGlobalMute();

Routine	Required Header	Distribution
gvGetGlobalMute	<gv.h>	SDKZIP

Return Value

GVTrue if global mute turned on, GVFalse if off.

Section Reference: [Gamespy Voice SDK](#)

See Also: [gvSetGlobalMute](#)

gvGetPushToTalk

Tells you if PushToTalk is currently turned on or off.

```
GVBool gvGetPushToTalk(  
    GVDevice device );
```

Routine	Required Header	Distribution
gvGetPushToTalk	<gv.h>	SDKZIP

Return Value

GVTrue if turned on, GVFalse if turned off.

Parameters

device

[in] The handle to the capture device.

Section Reference: [Gamespy Voice SDK](#)

See Also: [gvSetPushToTalk](#), [gvSetCaptureMode](#)

gvIsDeviceStarted

Checks to see if a whether or not a device has been started as the given device type.

```
GVBool gvIsDeviceStarted(  
    GVDevice device,  
    GVDeviceType type );
```

Routine	Required Header	Distribution
gvIsDeviceStarted	<gv.h>	SDKZIP

Return Value

Returns GVTrue if the device has been started, GVFalse if not.

Parameters

device

[in] The handle to the device.

type

[in] Specifies capture or playback device.

Section Reference: [Gamespy Voice SDK](#)

See Also: [gvStartDevice](#), [gvStopDevice](#)

gvlsSourceTalking

Determines if a particular source is currently talking on the specified device.

```
GVBool gvlsSourceTalking(  
    GVDevice device,  
    GVSource source );
```

Routine	Required Header	Distribution
gvlsSourceTalking	<gv.h>	SDKZIP

Return Value

Returns GVTrue if the source is talking on the specified device; otherwise, GVFalse.

Parameters

device

[in] The handle to the device.

source

[in] The source identifier.

Section Reference: [Gamespy Voice SDK](#)

See Also: [gvListTalkingSources](#)

gvListDevices

Gets a list of devices available on the system.

```
int gvListDevices(  
    GVDeviceInfo devices[],  
    int maxDevices,  
    GVDeviceType types );
```

Routine	Required Header	Distribution
gvListDevices	<gv.h>	SDKZIP

Return Value

Returns the number of devices that it put into the list. Return value of 0 may indicate an error or that no devices were found.

Parameters

devices

[out] The list of device details to be filled in by the function.

maxDevices

[in] The number of elements in the devices array.

types

[in] The types of devices to survey.

Remarks

You can request capture devices with `GV_CAPTURE`, playback devices with `GV_PLAYBACK`, or capture and playback devices with `GV_CAPTURE_AND_PLAYBACK`. For `GV_CAPTURE_AND_PLAYBACK`, it can list capture devices, playback devices, and devices that support both capture and playback.

Section Reference: [Gamespy Voice SDK](#)

See Also: [GVDeviceInfo](#)

gvListTalkingSources

Gets a list of all of the sources that are currently talking on a particular device.

```
int gvListTalkingSources(  
    GVDevice device,  
    GVSource sources[],  
    int maxSources );
```

Routine	Required Header	Distribution
gvListTalkingSources	<gv.h>	SDKZIP

Return Value

Returns the number of sources that are talking on the device.

Parameters

device

[in] The handle of the device.

sources

[out] An array to receive the sources, filled in by the function.

maxSources

[in] The number of elements in the sources array.

Remarks

The function will return the number of sources that were talking on the device, and it will store their GVSources in the sources array. 0 will be returned if there are no sources talking.

GV has a hardcoded limit that does not allow more than 8 sources to talk simultaneously. This allows it to preallocate memory that it needs to store for a source while it is talking. A user will typically not understand more than 2 or 3 users talking simultaneously, so the limit should be high enough. If the application attempts to play audio from more than 8 sources at a time, audio for the 9th source will be automatically dropped.

Section Reference: [Gamespy Voice SDK](#)

See Also: [gvIsSourceTalking](#)

gvNewCustomDevice

Creates a custom device, which allows an application to supply its own audio hardware interface.

```
GVDevice gvNewCustomDevice(  
    GVDeviceType type );
```

Routine	Required Header	Distribution
gvNewCustomDevice	<gv.h>	SDKZIP

Return Value

Returns a handle to the new custom device if successful; NULL if it cannot create the device.

Parameters

type

[in] Specifies whether device handles capture or playback or both

Section Reference: [Gamespy Voice SDK](#)

See Also: [gvGetCustomPlaybackAudio](#)

gvNewDevice

Initializes a device.

```
GVDevice gvNewDevice(  
    GVDeviceID deviceId,  
    GVDeviceType type );
```

Routine	Required Header	Distribution
gvNewDevice	<gv.h>	SDKZIP

Return Value

If the device was successfully initialized, a handle to the device will be returned. If there was an error setting up the device, NULL will be returned.

Parameters

deviceID

[in] The ID for the device to be initialized

type

[in] Specifies whether device handles capture or playback or both

Remarks

A device that supports both capture and playback may be initialized for just one or the other or both.

Section Reference: [Gamespy Voice SDK](#)

See Also: [gvFreeDevice](#), [gvNewCustomDevice](#)

gvPlayPacket

Plays a packet retrieved from the capture buffer.

```
void gvPlayPacket(  
    GVDevice device,  
    const GVByte * packet,  
    int len,  
    GVSource source,  
    GVFrameStamp frameStamp,  
    GVBool mute );
```

Routine	Required Header	Distribution
gvPlayPacket	<gv.h>	SDKZIP

Parameters

device

[in] The handle to the playback device.

packet

[in] The packet with audio data.

len

[in] The packet's length.

source

[in] The source that originated the audio.

frameStamp

[in] The packet's frame stamp.

mute

[in] Mutes the packet - allows having a player muted, but keeping track of the fact that the source is really speaking

Remarks

GV will schedule the packet to be played soon. A short delay is added which enables the packets to be synchronized before they are played, allowing for variations in Internet transit time and application timing. The packet is synchronized based on its source, so you must ensure that each unique talker has his own unique source, and all packets are played using the correct source. Note that the same packet can be played on multiple playback devices.

The application must also call `gvThink` on a regular basis to ensure that the packets are actually played.

Section Reference: [Gamespy Voice SDK](#)

See Also: [gvCapturePacket](#), [gvThink](#)

gvRunSetupWizard

Interacts with the user to set up the capture and playback devices.

```
GVBool gvRunSetupWizard(  
    GVDeviceID captureDeviceID,  
    GVDeviceID playbackDeviceID );
```

Routine	Required Header	Distribution
gvRunSetupWizard	<gv.h>	SDKZIP

Return Value

Returns GVTrue if the user successfully completes the wizard, GVFalse otherwise.

Parameters

captureDeviceID

[in] Id of the capture device to set up

playbackDeviceID

[in] Id of the playback device to set up

Remarks

For use with Win32 only, if the user has DirectX 8 or greater. The wizard will take over control of the program while it executes. It has the user speak into the capture device, and monitors the audio to automatically set system level capture and playback volumes. **gvRunSetupWizard** stores setup information in the registry.

Section Reference: [Gamespy Voice SDK](#)

See Also: [gvAreDevicesSetup](#)

gvSetCaptureMode

Sets the capture mode for the device.

```
void gvSetCaptureMode(  
    GVDevice device,  
    GVCaptureMode captureMode );
```

Routine	Required Header	Distribution
gvSetCaptureMode	<gv.h>	SDKZIP

Parameters

device

[in] The handle to the capture device.

captureMode

[in] The new capture mode.

Remarks

The default mode for the SDK is `GVCaptureModeThreshold`. In `GVCaptureModeThreshold`, a capture device is on and captures speech based on the current threshold value. When you change to `GVCaptureModePushToTalk`, the SDK will save the current Threshold value, set the threshold value to 0, and stop the capture device. This mode also allows use of the following functions: `gvSetPushToTalk()`, `gvGetPushToTalk()`. When you switch the captureMode to `GVCaptureModeThreshold`, the saved Threshold value will be restored and the capture device will be started. If the device is not a capture device, **`gvSetCaptureMode()`** will assert.

Section Reference: [Gamespy Voice SDK](#)

See Also: [gvGetCaptureMode](#), [GVCaptureMode](#)

gvSetCaptureThreshold

Sets the threshold volume on a device. A packet will only be passed to the application if its peak volume is at least as high as the capture threshold.

```
void gvSetCaptureThreshold(  
    GVDevice device,  
    GVScalar threshold );
```

Routine	Required Header	Distribution
gvSetCaptureThreshold	<gv.h>	SDKZIP

Parameters

device

[in] The handle to the device.

threshold

[in] The threshold volume

Remarks

The range for threshold is 0.0 to 1.0. A value of approximately 0.10 to 0.15 will generally work well, although ideally the user should have a way to configure the threshold. GV will continue passing packets to the application for about half a second after the peak volume drops below the threshold. This helps to catch speech in which the level trails off or has a small dip. The default threshold is 0.0, which means that all audio will be considered over the threshold and will be captured. To remove a threshold that has been set, simply call this function again with a threshold of 0.0.

Section Reference: [Gamespy Voice SDK](#)

See Also: [gvGetCaptureThreshold](#)

gvSetCodec

Sets the codec to be used by the SDK.

```
GVBool gvSetCodec(  
    GVCodec codec );
```

Routine	Required Header	Distribution
gvSetCodec	<gv.h>	SDKZIP

Return Value

Returns GVTrue if successful; otherwise, GVFalse.

Parameters

codec

[in] The codec identifier.

Remarks

The first thing to do after initializing the SDK is to set the codec you would like to use. The codec cannot be changed while any devices are initialized, so an application will typically just set the codec once, when it starts using voice.

The codec must be one of the following values:

GVCCodecSuperHighQuality

GVCCodecHighQuality

GVCCodecAverage

GVCCodecLowBandwidth

GVCCodecSuperLowBandwidth.

Section Reference: [Gamespy Voice SDK](#)

See Also: [gvGetCodecInfo](#), [gvSetCustomCodec](#)

gvSetCustomCaptureAudio

For a custom capture device, encodes captured audio from a stream into a packet, storing it at provided memory.

```
GVBool gvSetCustomCaptureAudio(  
    GVDevice device,  
    const GVSample * audio,  
    int numSamples,  
    GVByte * packet,  
    int * packetLen,  
    GVFrameStamp * frameStamp,  
    GVScalar * volume );
```

Routine	Required Header	Distribution
gvSetCustomCaptureAudio	<gv.h>	SDKZIP

Return Value

Returns GVTrue if the function successfully encodes the audio into the packet, otherwise GVFalse. GVFalse will be returned if a threshold is set and the audio's peak volume did not cross the threshold.

Parameters

device

[in] The handle to the custom capture device.

audio

[in] The incoming audio stream.

numSamples

[in] The number of samples to capture.

packet

[out] The memory location where the packet will be stored.

packetLen

[ref] The number of bytes available in the packet,

frameStamp

[out] Receives the frame stamp for the packet.

volume

[out] Receives the peak volume for the packet.

Remarks

The numSamples parameter must be a multiple of the codec's samplesPerFrame; this ensures that no data needs to be buffered by the SDK.

Section Reference: [Gamespy Voice SDK](#)

See Also: [gvGetCodecInfo](#)

gvSetCustomCodec

Tells GV to use an application-provided codec instead of a built-in codec.

```
void gvSetCustomCodec(  
    GVCustomCodecInfo * info );
```

Routine	Required Header	Distribution
gvSetCustomCodec	<gv.h>	SDKZIP

Parameters

info

[in] The application fills in this structure with the information that the SDK needs to use the custom codec.

Remarks

The first thing to do after initializing the SDK is to set the codec you would like to use. The codec cannot be changed while any devices are initialized, so an application will typically just set the codec once, when it starts using voice.

Before calling this function, the application must fill in the `GVCustomCodecInfo` structure with information about the codec to be used.

`m_samplesPerFrame` is the number of samples that this codec expects in a raw (unencoded) frame of audio. This can be whatever value is used by the codec, however it is typically about 160 samples.

`m_encodedFrameSize` is the number of bytes in an encoded frame of audio produced by this codec. The ratio of the samples per frame and encoded frame size is directly related to the codec's output stream bit rate.

`m_newDecoderCallback` is used to allocate a new decoder instance for each incoming source. Some codecs do not require this, and they should set the `m_newDecoderCallback` member to `NULL`. For codecs that do require per-source data, they should allocate a new decoder data state and set the data parameter to point to it, then return `TRUE`. If they cannot allocate a new decoder data, then they should return `GVFalse`.

The `m_freeDecoderCallback` is used to free decoder data allocated through `m_newDecoderCallback`. If a codec set `m_newDecoderCallback` to `NULL`, it should set `m_freeDecoderCallback` to `NULL` as well. Otherwise it should provide a function that frees the decoder data.

`m_encodeCallback` is used to encode data. The `in` parameter points to the input data, with `is` a single raw frame of samples. The number of samples passed to this function will always be `m_samplesPerFrame`. The `out` parameter points to the memory into which the callback should decode the input data. The memory will always be large enough to hold one frame of compressed audio, which will always be `m_encodedFrameSize` bytes long.

`m_decodeCallback` is used to decode data. The `in` parameter will point to an encoded frame of audio, which will be `m_encodedFrameSize` bytes long. The `out` parameter which will be large enough to hold `m_samplesPerFrame` samples of audio. The decoder data is provided for codecs that need it. The important thing to know with the decode callback is that it should not decode directly into the out buffer, but it should add to it. This allows GV to decode and mix at the same time, without having to decode into a temporary buffer which would then be mixed into the output stream.

Section Reference: [Gamespy Voice SDK](#)

See Also: [gvSetCodec](#)

gvSetDeviceVolume

Sets a device's volume.

```
void gvSetDeviceVolume(  
    GVDevice device,  
    GVDeviceType type,  
    GVScalar volume );
```

Routine	Required Header	Distribution
gvSetDeviceVolume	<gv.h>	SDKZIP

Parameters

device

[in] The handle to the device.

type

[in] Specifies setting the capture volume, playback volume, or both.

volume

[in] The volume, ranging from 0.0 to 1.0.

Section Reference: [Gamespy Voice SDK](#)

See Also: [gvGetDeviceVolume](#)

gvSetFilter

Sets a device's filter callback.

```
void gvSetFilter(  
    GVDevice device,  
    GVDeviceType type,  
    gvFilterCallback callback );
```

Routine	Required Header	Distribution
gvSetFilter	<gv.h>	SDKZIP

Parameters

device

[in] The handle to the device.

type

[in] Set the filter on capture, playback, or both.

callback

[in] The filter callback to use.

Remarks

Filtering allows you to process, or just monitor, audio that has been captured or is being played. A filter callback, prototyped as the `gvFilterCallback` type, is passed the device the filtering is happening on, the audio to filter, and the audio's frame stamp. The audio will always be a single raw frame of audio. Use `gvGetCodecInfo` to get the number of samples in a raw frame.

The callback can modify the audio in any way that it wants. However once the function returns it can no longer access the audio. For capture devices, audio will only be passed to the filter if it crosses the threshold (if one is set). For playback devices, audio is filtered after all of the sources have been mixed.

You can use **`gvSetFilter`** to set a filter on any device, and to set it for capture or playback. A device can have only one capture filter at a time and only one playback filter at a time. To clear a filter, call this function with the callback set to `NULL`.

Section Reference: [Gamespy Voice SDK](#)

See Also: [gvFilterCallback](#)

gvSetGlobalMute

Sets the global mute value - defaults to false.

```
void gvSetGlobalMute(  
    GVBool mute );
```

Routine	Required Header	Distribution
gvSetGlobalMute	<gv.h>	SDKZIP

Parameters

mute

[in] Set to GVTrue to globally mute all play packets.

Remarks

When **gvSetGlobalMute** mute is true, all data passed to gvPlayPacket will not be played on the playback device. You will still be able to check the gvIsSourceTalking to know that you are sending your voice packets to play. When **gvSetGlobalMute** is false, all gvPlayPacket data will be played, if the gvPlayPacket mute is false.

Section Reference: [Gamespy Voice SDK](#)

See Also: [gvGetGlobalMute](#), [gvPlayPacket](#)

gvSetPushToTalk

Used to turn on or off capturing for a device. Must be in GVCaptureModePushToTalk mode.

```
void gvSetPushToTalk(  
    GVDevice device,  
    GVBool talkOn );
```

Routine	Required Header	Distribution
gvSetPushToTalk	<gv.h>	SDKZIP

Parameters

device

[in] The handle to the capture device.

talkOn

[in] GVTrue to start capture device and capture speech, GVFalse to turn off capture device.

Remarks

When called with talkOn true, the device will start the capture device and capture all speech. When set to false, the capture device is turned off.

Section Reference: [Gamespy Voice SDK](#)

See Also: [gvGetPushToTalk](#), [gvSetCaptureMode](#)

gvSetUnpluggedCallback

Sets a callback to be called when the SDK detects that the device was unplugged or is no longer functioning.

```
void gvSetUnpluggedCallback(  
    GVDevice device,  
    gvUnpluggedCallback unpluggedCallback );
```

Routine	Required Header	Distribution
gvSetUnpluggedCallback	<gv.h>	SDKZIP

Parameters

device

[in] The handle to the device.

unpluggedCallback

[in] The callback to set. Can be NULL.

Remarks

A `gvUnpluggedCallback` allows an application to know if a device is unplugged or otherwise stops working. The callback will be called when the SDK detects that the device has been unplugged. The device will be freed by the SDK immediately after the callback returns and cannot be used again by the application.

Section Reference: [Gamespy Voice SDK](#)

See Also: [gvUnpluggedCallback](#)

gvStartDevice

Starts a device capturing and/or playing audio.

```
GVBool gvStartDevice(  
    GVDevice device,  
    GVDeviceType type );
```

Routine	Required Header	Distribution
gvStartDevice	<gv.h>	SDKZIP

Return Value

GVTrue if the device was started successfully.

Parameters

device

[in] The handle to the device.

type

[in] Specifies capture, playback, or both.

Remarks

Once a device has been initialized, it is ready to start capturing or playing audio. After a capture device is started, it will begin capturing audio and passing it back to the application. After a playback device is started, it will play any audio that the application passes it. To start a device, use **gvStartDevice**.

The device parameter is the handle of the device to start. The type parameter specifies if the device should start capturing (GV_CAPTURE), playing (GV_PLAYBACK), or capturing and playing (GV_CAPTURE_AND_PLAYBACK). For devices that support both capture and playback, each can be started independently. The function will return GVTrue if the device was started successfully, and it will return GVFalse if there was an error.

Section Reference: [Gamespy Voice SDK](#)

See Also: [gvStopDevice](#), [gvIsDeviceStarted](#)

gvStartup

Initializes the SDK.

```
GVBool gvStartup(  
    HWND hWnd );
```

Routine	Required Header	Distribution
gvStartup	<gv.h>	SDKZIP

Return Value

Returns GVTrue if the SDK was able to startup successfully.

Parameters

hWnd

[in] Handle to the application's main window. [Win32 only]

Remarks

Before doing anything else with GV, you must call **gvStartup**. The function does any necessary internal initialization. It will return GVFalse in case of an error initializing. The HWND passed to the Win32 version is the handle for the application's main window. This can be NULL if the application does not have a main window.

Section Reference: [Gamespy Voice SDK](#)

See Also: [gvCleanup](#)

gvStopDevice

Stops a device that is capturing and/or playing audio.

```
void gvStopDevice(  
    GVDevice device,  
    GVDeviceType type );
```

Routine	Required Header	Distribution
gvStopDevice	<gv.h>	SDKZIP

Parameters

device

[in] The handle to the device.

type

[in] Specifies capture, playback, or both.

Remarks

When you want a device to stop capturing or playing, use **gvStopDevice**. When a capture device is stopped, it will stop passing captured audio to the application. When a playback device is stopped, it will stop playing audio. For devices that support both capture and playback, each can be stopped independently.

Section Reference: [Gamespy Voice SDK](#)

See Also: [gvStartDevice](#), [gvIsDeviceStarted](#)

gvThink

Allows playback devices to play audio scheduled for playback.

void gvThink();

Routine	Required Header	Distribution
gvThink	<gv.h>	SDKZIP

Remarks

`gvPlayPacket` only schedules a packet to be played in the future. The application must also call **gvThink** on a regular basis to ensure that the packets are actually played.

gvThink will check, for each device, how much space has become available for writing in the playback buffer (which may or may not be on the actual sound hardware). It will then check to see if the device has any sources that have audio which should be played during the time period that the newly available space represents. If so, the audio will be mixed into the playback buffer, and the audio will then be played when the playback position reaches that point in the buffer. If the playback device is stopped before that happens, then the audio will not be played, even if the device is then restarted.

gvThink should generally be called once for each run through the application's main loop, or approximately every 10-30ms. If it is not called often enough, the playback position will reach a point in the playback buffer that GV has not yet had a chance to mix to, resulting in an audible skipping effect.

Section Reference: [Gamespy Voice SDK](#)

See Also: [gvPlayPacket](#)

Voice SDK Callbacks

[gvFilterCallback](#)

Used to filter audio that has either been captured or is about to be played. Can also be used to monitor audio.

[gvUnpluggedCallback](#)

Called when a device has been unplugged.

gvFilterCallback

Used to filter audio that has either been captured or is about to be played. Can also be used to monitor audio.

```
typedef void (*gvFilterCallback)(  
    GVDevice device,  
    GVSample * audio,  
    GVFrameStamp frameStamp );
```

Routine	Required Header	Distribution
gvFilterCallback	<gv.h>	SDKZIP

Parameters

device

[in] The handle to the device.

audio

[ref] A frame of audio to be filtered.

frameStamp

[in] The framestamp for the frame of audio.

Remarks

Filtering allows you to process, or just monitor, audio that has been captured or is being played. The audio will always be a single raw frame of audio. Use `gvGetCodecInfo` to get the number of samples in a raw frame.

The callback can modify the audio in any way that it wants. However once the function returns it can no longer access the audio. For capture devices, audio will only be passed to the filter if it crosses the threshold (if one is set). For playback devices, audio is filtered after all of the sources have been mixed.

Section Reference: [Gamespy Voice SDK](#)

See Also: [gvSetFilter](#)

gvUnpluggedCallback

Called when a device has been unplugged.

```
typedef void (*gvUnpluggedCallback)(  
    GVDevice device );
```

Routine	Required Header	Distribution
gvUnpluggedCallback	<gv.h>	SDKZIP

Parameters

device

[in] The handle to the device.

Remarks

A **gvUnpluggedCallback** allows an application to know if a device is unplugged or otherwise stops working. The callback will be called when the SDK detects that the device has been unplugged. The device will be freed by the SDK immediately after the callback returns and cannot be used again by the application.

Section Reference: [Gamespy Voice SDK](#)

See Also: [gvSetUnpluggedCallback](#)

Voice SDK Structures

GVCustomCodecInfo	Information to define a custom codec.
GVDeviceInfo	Information for an audio device

GVCustomCodecInfo

Information to define a custom codec.

typedef struct

{

int *m_samplesPerFrame*;

int *m_encodedFrameSize*;

GVBool (*)(GVDecoderData * **data**) *m_newDecoderCallback*;

void (*)(GVDecoderData **data**) *m_freeDecoderCallback*;

void (*)(GVByte * **out**, **const GVSample * in**) *m_encodeCallback*;

void (*)(GVSample * **out**, **const GVByte * in**, GVDecoderData **data**) *m_decodeCallback*;

} ***GVCustomCodecInfo***;

Members

m_samplesPerFrame

Number of samples in an unencoded frame.

m_encodedFrameSize

Number of bytes in an encoded frame.

m_newDecoderCallback

Used to allocate a new decoder instance for each incoming source.

m_freeDecoderCallback

Used to free decoder data allocated through *m_newDecoderCallback*.

m_encodeCallback

Used to encode data.

m_decodeCallback

Called to decode data. Decode must **add to**, not set the output.

Section Reference: [Gamespy Voice SDK](#)

See Also: [gvSetCustomCodec](#)

GVDeviceInfo

Information for an audio device.

```
typedef struct  
{  
    GVDeviceID m_id;  
    char m_name[GV_DEVICE_NAME_LEN];  
    GVDeviceType m_deviceType;  
    GVDeviceType m_defaultDevice;  
    GVHardwareType m_hardwareType;  
} GVDeviceInfo;
```

Members

m_id

Used if you initialize this device with `gvNewDevice`.

m_name

A user-readable name for the device.

m_deviceType

Indicates if this device is for capture, playback, or both capture and playback.

m_defaultDevice

Indicates if this device is the default capture device, default playback device, both, or neither. If neither, the value will be 0 (this will always be the case on the PS2).

m_hardwareType

More information about the device's actual hardware. Will differ based on platform. See `GVHardwareType` for settings.

Section Reference: [Gamespy Voice SDK](#)

See Also: [gvListDevices](#), [GVHardwareType](#)

Voice SDK Enumerations

GVCaptureMode	enums used with gvSetCaptureMode() and gvGetCaptureMode().
GVCodec	Identifies each of the default codecs available.
GVHardwareType	Hardware type of a device.

GVCaptureMode

enums used with `gvSetCaptureMode()` and `gvGetCaptureMode()`.

```
typedef enum  
{  
    GVCaptureModeThreshold,  
    GVCaptureModePushToTalk  
} GVCaptureMode;
```

Constants

GVCaptureModeThreshold

mode captures speech based on the current threshold value.

GVCaptureModePushToTalk

mode captures speech when gvSetPushToTalk is turned on.

Remarks

The default mode for the SDK is **GVCaptureModeThreshold**. In **GVCaptureModeThreshold**, a capture device is on and captures speech based on the current threshold value. When you change to **GVCaptureModePushToTalk**, the SDK will save the current Threshold value, set the threshold value to 0, and stop the capture device. This mode also allows use of the following functions: `gvSetPushToTalk()`, `gvGetPushToTalk()`. When you switch the captureMode to **GVCaptureModeThreshold**, the saved Threshold value will be restored and the capture device will be started. If the device is not a capture device, `gvSetCaptureMode()` will assert.

Section Reference: [Gamespy Voice SDK](#)

See Also: [gvSetCaptureMode](#), [gvGetCaptureMode](#)

GVCCodec

Identifies each of the default codecs available.

```
typedef enum  
{  
    GVCCodecSuperHighQuality,  
    GVCCodecHighQuality,  
    GVCCodecAverage,  
    GVCCodecLowBandwidth,  
    GVCCodecSuperLowBandwidth  
} GVCCodec;
```

Remarks

The codecs are arranged in order of descending quality and bandwidth. In other words, the codecs higher up on the list are of higher audio quality and use more bandwidth, while the codecs lower on the list are of lower audio quality and use less bandwidth.

The **GVCodecAverage** codec produces good quality audio with a reasonable bandwidth cost. It is generally the best codec to use, and you should only use another codec if you are restricted to lower bandwidth or need high quality audio.

The particular stats for a codec can be obtained using `gvGetCodecInfo`.

Section Reference: [Gamespy Voice SDK](#)

See Also: [gvGetCodecInfo](#)

GVHardwareType

Hardware type of a device.

typedef enum

{

GVHardwareDirectSound,

GVHardwarePS2Spu2,

GVHardwarePS2Headset,

GVHardwarePS2Eyeto,

GVHardwareCustom

} GVHardwareType;

Constants

GVHardwareDirectSound

The hardware type for Win32 devices.

GVHardwarePS2Spu2

SPU2 on the PS2.

GVHardwarePS2Headset

USB headsets on the PS2.

GVHardwarePS2Eyeto

Eyeto device on the PS2.

GVHardwareCustom

Custom hardware type.

Section Reference: [Gamespy Voice SDK](#)

Persistent Storage SDK

Overview

The GameSpy Persistent Storage SDK allows a developer to associate any data they want with a player profile and have it stored on a secure central server. Both binary and ASCII data can be used, and there is no fixed size limit to the amount of data that can be stored. The Persistent Storage SDK is designed to allow many possible uses including: secure storage of character data, player settings / configuration, ranking / ladder information, and personal player information (home page / clan, etc).

Backend processes can also access the data to dynamically update a player's ranking, ladder, or tournament information. No fixed data structures are imposed, and developers are allowed to use the storage space in any reasonable manner. Access to the data is securely controlled through either personal logins (Presence and Messaging SDK) or unique CD Keys (CD Key SDK).

The Presence and Messaging SDK allows each account to have multiple profiles, and each CD Key can have multiple profiles associated with it via nicknames. Each profile can have multiple "data records" associated with it (for example, to store multiple character records separately), and each record can have 4 different types of data associated with it:

- Private Read/Write Data
- Private Read-Only Data
- Public Read/Write Data
- Public Read-Only Data

Private data can only be accessed by the authenticated user whose profile it is associated with. This is typically used to store player configuration/settings, private character data, and favorites. Public data can be accessed by any other player, and is typically used for things like ranking and personal information (home page, clan affiliation, etc). Read/Write data can be updated at-will by the client that owns the profile. Read-Only data can only be updated by a process running on our

backend, for example, in conjunction with the Stats and Tracking SDK to update ladder and ranking information.

The Persistent Storage SDK is built on top of the Stats and Tracking SDK, and uses much of the same code and terminology. However, you can choose to implement Persistent Storage without Stats and Tracking (or vice-versa).

Data Storage

Data storage records are keyed off of the combination of profileid and index. In each record there are four separate bins with separate access control. For CD Key authentication, the profileid is unique based on the CD Key and nickname. A function is included to lookup the profileid based on the CD Key Hash and the nick. When you request to get or set data, you pass in the profileid, index, and bin type.

Although you can have multiple indexes of data per profile, we recommend that you try to store all your data in index 0. It's better to have more keys\values (or a single, large binary structure) in a single record than having a bunch of small key\value sets or binary structures under different indexes in the database. If you do choose to use multiple indexes, you are not constrained to using consecutive numbers (if you don't need to). You can use any integer as the index value.

You have two choices for the format in which to save data. You can save it in our standard key\value delimited ASCII format, or a custom format of your choosing (binary or ASCII).

The advantages and disadvantages are as follows:

ASCII key\value format:

- Easy to parse / extend without worrying about versioning of the data
- Special functions included in the SDK to retrieve and update subsets of keys, meaning you don't have to get the entire data bin to update only a single key.
- Can often be larger (data-size-wise) than a fixed binary format

Binary / Custom format:

- Most efficient use of space
- Can dump existing game structures or save-game formats easily (without converting them to key\value pairs)
- Need to worry about byte-order and platform / version issues

If you use the key\value format, all data should be in the form of key\value pairs. A data set consists of key\value pairs, beginning with the '\ ' character, and ending with the last value. For example:

```
" \key1\value1\mykeyname\mykeyvalue\keyhere\valueher
```

Binary or custom formats don't have a fixed spec - they are treated as raw blocks of data. You simply request the whole block and get a pointer with a length, and set the whole block by passing in a pointer with the length.

Authentication

The Persistent Storage SDK does authentication in a unique two-part fashion that allows for the highest level of flexibility in the SDK implementation.

For most games, it will be appropriate to use the Persistent Storage SDK on each client - for example, to allow a client to update their own information, or query for other players' information. In this case, the authentication process is straight-forward:

1. Connect to the Persistent Storage Server using `InitStatsConnection()`
2. Call `GetChallenge()` to get the challenge string to use for authentication
3. Call `GenerateAuth()` with the challenge string and plain-text password or CD Key to generate the validation token
4. Call `PreAuthenticatePlayerCD` or `PreAuthenticatePlayerPM` to authenticate the player, allowing them to query for their private data, and update their read-write data.

For other games, it may be better to have all communications with the Persistent Storage Server done by the multiplayer game host (server). In this scenario, the server will need to authenticate each client (so that it can read their private data, and update their read-write data), but this needs to be done in a manner whereby the server never sees the plaintext passwords or CD Keys of their clients. To do this, the server needs to implement the two-part authentication:

1. On server startup, connect to the Persistent Storage Server using `InitStatsConnection()`
2. When a client connects, call `GetChallenge()` and send that challenge string to the client
3. On the client side, take the challenge string received from the game server, and use the `GenerateAuth()` function with the plain-text password or CD Key. `GenerateAuth()` will return a validation token which is NOT reversible to determine the password or CD Key

4. Send the validation token, along with the login information (profileid or CD Key Hash) to the game server
5. On the game server, call `PreAuthenticatePlayerCD()` or `PreAuthenticatePlayerPM()` to authenticate the player
6. Repeat the authentication process for each client as they connect to make sure they are all authenticated

If you are implementing the Stats & Tracking SDK as well, you'll note that the second process describes exactly what must be done to get authentication data for clients to include in the stats snapshot. The validation token used with `PreAuthenticatePlayer` is the same value that should be included in the "auth_N" key for that player. See the Stats & Tracking SDK for more details.

Note that you do NOT need to authenticate a client to just read public values. You can easily implement "guest" players in this fashion, who do not need to provide login or CD Key information - they can still see data for other users, they simply cannot save data for themselves. Just call `InitStatsConnection()` and use the `GetPersistData()` and `GetPersistDataValues()` functions directly.

File Manifest

The following files should be included with this package. If any of the files are missing, please [contact devsupport@gamespy.com](mailto:devsupport@gamespy.com).

File	Description
<i>gstats.c,h</i>	Stats and Tracking header file and code
<i>statstest.c</i> SDK	Example and test code for the Stats / Tracking SDK
<i>gstats.dsw</i>	Devstudio project for SDK / sample code
<i>gpersist.h</i>	Persistent Storage SDK header
<i>\persisttest\</i> SDK	Example and test code for Persistent Storage SDK
<i>md5c.c, md5.h</i>	MD5 hash code and header
<i>gbucket.c,h</i>	Bucket helper code and header
<i>nonport.c,h</i>	Platform-specific code
<i>darray.c,h</i>	Code for managing dynamic arrays
<i>hashtable.c,h</i>	Hash table code and header

Implementation

The following is a quick rundown of the basic steps needed to support the SDK. The `gpersist.h` file contains much more extensive documentation for each function. The `persisttest.c` sample contains sample usage of all of these functions.

Step 1: Initialize the Stats / Persistent Storage Server Connection

Before calling any of the actual Persistent Storage functions, you'll need to connect to the Persistent Storage server.

First, set the global `gcd_gamename` and `gcd_secret_key` variables to your gamename and secret key. If these values aren't set correctly, you will be unable to connect to the persistent storage server.

Once these values are set, call

```
int InitStatsConnection(int gameport)
```

...with the game port that the host is running on. If your game doesn't use multiple ports (or doesn't support more than one host per machine) then you can just use 0.

This call is blocking and may take 1-2 seconds to complete the authentication process. This is the only blocking call in the SDK; all other calls will return immediately. When you are done with the Persistent Storage functions you can call

```
void CloseStatsConnection(void);
```

Step 2: Authenticate Player

Before you can request private records, or set any read/write records, you need to authenticate the player. If you are just reading public records you do not need to authenticate. This is done using the two-step process described in the Authentication section of this document. The calls you

will need to use are:

```
char *GetChallenge(statsgame_t game);
char *GenerateAuth(char *challenge, char *password,
void PreAuthenticatePlayerPM(int localid, int profileid, char *password, void *callback);
void PreAuthenticatePlayerCD(int localid, char *nick, char *password, void *callback);
void PreAuthenticatePlayerPartner(int localid, const char *challenge, char *password, void *callback);
```

You first call [GetChallenge](#) to get the challenge value used for authentication. If you aren't using the Stats and Tracking SDK, just pass in [NULL](#) for game. This challenge value is then passed along with the [password](#) (either a profile password or an unhashed CD Key or the partner challenge) to the [GenerateAuth](#) function to create a validation token (response). This validation token is then used with either [PreAuthenticatePlayerPM](#) or [PreAuthenticatePlayerCD](#) or [PreAuthenticatePlayerPartner](#) to begin the authentication process.

The callback specified as "callback" in those functions will be called when the authentication is complete (either successful or not).

Step 3: Get / Set Data

Getting / Setting of data is done through four functions.

```
void GetPersistData(int localid, int profileid, pers_data_t *data, void *callback);
void SetPersistData(int localid, int profileid, pers_data_t *data, void *callback);
void GetPersistDataValues(int localid, int profileid, const char *keys, pers_data_t *data, void *callback);
void SetPersistDataValues(int localid, int profileid, const char *keys, pers_data_t *data, void *callback);
```

[GetPersistData](#) / [SetPersistData](#) work with the entire record as a binary blob. When you call Get, you receive the entire data block, and when you call Set, the entire data block is replaced with your new data.

[GetPersistDataValues](#) and [SetPersistDataValues](#) are designed to work with data stored in ASCII key/value format. You can pass a set of keys to [GetPersistDataValues](#) to only get a subset of the data stored in the record, and when you pass key/value pairs to

[SetPersistDataValues](#) only those values are updated - any other existing keys in the record are maintained.

Step 4: Think

You need to call the [PersistThink](#) function any time an asynchronous operation is in that you call this in your main loop at all times while you are connected to the stats server, so that if the stats server disconnects it can be detected immediately.

Persistent Storage SDK Functions

CloseStatsConnection	Closes the connection to the stats server. You should do this when done with the connection.
GenerateAuth	Create a validation token (response) for use when beginning the authentication process.
GetChallenge	Get the challenge value used for authentication.
GetPersistData	Gets the entire block of persistent data for a user.
GetPersistDataModified	Gets the entire block of persistent data for a user, if it has been modified since the time provided.
GetPersistDataValues	Retrieves a subset of the data that is stored in key/value delimited pairs.
GetPersistDataValuesModified	Retrieves a subset of the data that is stored in key/value delimited pairs, but only if it has been modified since the time provided.
GetProfileIDFromCD	Given a nickname and CD Key hash, this will lookup the profileid for the user.

InitStatsConnection	Connects to the Persistent Storage server.
IsStatsConnected	Reports whether or not you are currently connected to the stats server.
PersistThink	Allows Persistent Storage SDK to continue internal processing including processing query replies. Also tests connection to the stats server.
PreAuthenticatePlayerCD	Authenticates a player on the Stats server.
PreAuthenticatePlayerPM	Authenticates a player on the Stats server.
SetPersistData	Sets the entire block of persistent data for a user.
SetPersistDataValues	If you are saving data in key\value delimited format, you can use this function to only set SOME of the key\value pairs. Only the key value pairs you include in they keyvalues parameter will be updated, the other pairs will stay the same.

CloseStatsConnection

Closes the connection to the stats server. You should do this when done with the connection.

void CloseStatsConnection();

Routine	Required Header	Distribution
CloseStatsConnection	<gpersist.h>	SDKZIP

Section Reference: [Gamespy Persistent Storage SDK](#)

See Also: [InitStatsConnection](#)

GenerateAuth

Create a validation token (response) for use when beginning the authentication process.

```
char * GenerateAuth(  
    char * challenge,  
    gsi_char * password,  
    char response[33] );
```

Routine	Required Header	Distribution
GenerateAuth	<gpersist.h>	SDKZIP

Return Value

A pointer to the output authentication string.

Parameters

challenge

[in] The string generated by GetChallenge()

password

[in] The CD Key (un-hashed) or profile password or partner challenge

response

[out] The output authentication string.

Remarks

Used to generate on the "challengeresponse" parameter for the PreAuthenticatePlayer functions.

Section Reference: [Gamespy Persistent Storage SDK](#)

See Also: [GetChallenge](#), [PreAuthenticatePlayerCD](#), [PreAuthenticatePlayerPM](#)

GetChallenge

Get the challenge value used for authentication.

```
char * GetChallenge(  
    statsgame_t game );
```

Routine	Required Header	Distribution
GetChallenge	<gpersist.h>	SDKZIP

Return Value

Returns a string to pass to `GenerateAuth` to create the authentication hash.

Parameters

game

[in] Your current game, or NULL if not using Stats and Tracking SDK

Section Reference: [Gamespy Persistent Storage SDK](#)

See Also: [GenerateAuth](#)

GetPersistData

Gets the entire block of persistent data for a user.

```
void GetPersistData(  
    int localid,  
    int profileid,  
    persisttype_t type,  
    int index,  
    PersDataCallbackFn callback,  
    void * instance );
```

Routine	Required Header	Distribution
GetPersistData	<gpersist.h>	SDKZIP

Parameters

localid

[in] Your game-specific reference number for this player

profileid

[in] The profileid of the player whose data you are getting.

type

[in] The type of persistent data you are getting.

index

[in] Each profile can have multiple persistent data records associated with them. Usually 0 is used.

callback

[in] Will be called with the data.

instance

[in] Pointer that will be passed to the callback function (for your use.)

Section Reference: [Gamespy Persistent Storage SDK](#)

GetPersistDataModified

Gets the entire block of persistent data for a user, if it has been modified since the time provided.

```
void GetPersistDataModified(  
    int localid,  
    int profileid,  
    persisttype_t type,  
    int index,  
    time_t modifiedsince,  
    PersDataCallbackFn callback,  
    void * instance );
```

Routine	Required Header	Distribution
GetPersistDataModified	<gpersist.h>	SDKZIP

Parameters

localid

[in] Your game-specific reference number for this player

profileid

[in] The profileid of the player whose data you are getting.

type

[in] The type of persistent data you are getting.

index

[in] Each profile can have multiple persistent data records associated with them. Usually 0 is used.

modifiedsince

[in] A time value to limit the request for data.

callback

[in] Will be called with the data.

instance

[in] Pointer that will be passed to the callback function (for your use.)

Remarks

Modification time is tracked for the given profileid/index, not on a per persisttype basis.

Data will only be returned if it has been modified since the time provided. If no data has been modified since that time, the callback will be called with a success value that indicates it is unmodified.

Section Reference: [Gamespy Persistent Storage SDK](#)

GetPersistDataValues

Retrieves a subset of the data that is stored in key\value delimited pairs.

```
void GetPersistDataValues(  
    int localid,  
    int profileid,  
    persisttype_t type,  
    int index,  
    gsi_char * keys,  
    PersDataCallbackFn callback,  
    void * instance );
```

Routine	Required Header	Distribution
GetPersistDataValues	<gpersist.h>	SDKZIP

Parameters

localid

[in] Your game-specific reference number for this player

profileid

[in] The profileid of the player whose data you are getting.

type

[in] The type of persistent data you are getting.

index

[in] Each profile can have multiple persistent data records associated with them. Usually 0 is used.

keys

[in] The key/value pairs to be updated.

callback

[in] Will be called with the data.

instance

[in] Pointer that will be passed to the callback function (for your use.)

Remarks

The data will be returned as a null-terminated string, If no data is available, len will be 0 in the callback.

To retrieve the entire data set, use GetPersistData.

Section Reference: [Gamespy Persistent Storage SDK](#)

GetPersistDataValuesModified

Retrieves a subset of the data that is stored in key\value delimited pairs, but only if it has been modified since the time provided.

```
void GetPersistDataValuesModified(  
    int localid,  
    int profileid,  
    persisttype_t type,  
    int index,  
    time_t modifiedsince,  
    gsi_char * keys,  
    PersDataCallbackFn callback,  
    void * instance );
```

Routine	Required Header	Distribution
GetPersistDataValuesModified	<gpersist.h>	SDKZIP

Parameters

localid

[in] Your game-specific reference number for this player

profileid

[in] The profileid of the player whose data you are getting.

type

[in] The type of persistent data you are getting.

index

[in] Each profile can have multiple persistent data records associated with them. Usually 0 is used.

modifiedsince

[in] A time value to limit the request for data.

keys

[in] The key/value pairs to be updated.

callback

[in] Will be called with the data.

instance

[in] Pointer that will be passed to the callback function (for your use.)

Remarks

Modification time is tracked for the given profileid/index, not on a per persisttype basis

Data will only be returned if it has been modified since the time provided. If no data has been modified since that time, the callback will be called with a success value that indicates it is unmodified.

The data will be returned as a null-terminated string, If no data is available, len will be 0 in the callback.

To retrieve the entire data set, use GetPersistData.

Section Reference: [Gamespy Persistent Storage SDK](#)

GetProfileIDFromCD

Given a nickname and CD Key hash, this will lookup the profileid for the user.

```
void GetProfileIDFromCD(  
    int localid,  
    gsi_char * nick,  
    char * keyhash,  
    ProfileCallbackFn callback,  
    void * instance );
```

Routine	Required Header	Distribution
GetProfileIDFromCD	<gpersist.h>	SDKZIP

Parameters

localid

[in] Your game-specific reference number for this player

nick

[in] The nickname of the user whose profileid you are looking up

keyhash

[in] The CD Key Hash of the user whose profileid you are looking up

callback

[in] Callback to be called when the lookup is completed.

instance

[in] Pointer that will be passed to the callback function (for your use.)

Remarks

If the user has never authenticated (and has no persistent data associated with them), the callback will indicate a failure. No persistent data can be retrieved for the user, since they don't have any stored. Persistent data can be stored, but only after authenticating with `PreAuthenticatePlayerCD`.

Section Reference: [Gamespy Persistent Storage SDK](#)

InitStatsConnection

Connects to the Persistent Storage server.

```
int InitStatsConnection(  
    int gameport );
```

Routine	Required Header	Distribution
InitStatsConnection	<gpersist.h>	SDKZIP

Return Value

Returns GE_NOERROR if connection succeeded, else one of the GE_ error codes. See Remarks.

Parameters

gameport

[in] The game port that the host is running on.

Remarks

If your game doesn't use multiple ports (or doesn't support more than one host per machine) then you can just use 0 for the gameport parameter.

If the connection fails, all Persistent Storage functions will fail.

Possible return errors include:

GE_NODNS: Unable to resolve stats server DNS

GE_NOSOCKET: Unable to create data socket

GE_NOCONNECT: Unable to connect to stats server

GE_DATAERROR: Unable to receive challenge from stats server, or bad challenge

GE_NOERROR: Connected to stats server and ready to send data.

Section Reference: [Gamespy Persistent Storage SDK](#)

See Also: [CloseStatsConnection](#)

IsStatsConnected

Reports whether or not you are currently connected to the stats server.

int IsStatsConnected();

Routine	Required Header	Distribution
IsStatsConnected	<gpersist.h>	SDKZIP

Return Value

1 if connected, 0 otherwise

Remarks

Even if your initial connection was successful, you may lose connection later and want to try to reconnect. If a callback returns unsuccessfully, check this function to see if it was because of a disconnection.

Section Reference: [Gamespy Persistent Storage SDK](#)

PersistThink

Allows Persistent Storage SDK to continue internal processing including processing query replies. Also tests connection to the stats server.

int PersistThink();

Routine	Required Header	Distribution
PersistThink	<gpersist.h>	SDKZIP

Return Value

0 if the connection to the stats server is lost, 1 otherwise.

Remarks

You need to call the **PersistThink** function any time an asynchronous operation is in progress. It will check for the incoming replies and call the callbacks associated with them as needed. It's recommended that you call this in your main loop at all times while you are connected to the stats server, so that if the stats server disconnects it can be detected immediately.

Section Reference: [Gamespy Persistent Storage SDK](#)

PreAuthenticatePlayerCD

Authenticates a player on the Stats server.

```
void PreAuthenticatePlayerCD(  
    int localid,  
    gsi_char * nick,  
    char * keyhash,  
    char * challengeresponse,  
    PersAuthCallbackFn callback,  
    void * instance );
```

Routine	Required Header	Distribution
PreAuthenticatePlayerCD	<gpersist.h>	SDKZIP

Parameters

localid

[in] Your game-specific reference number for this player

nick

[in] Nickname of the player to authenticate.

keyhash

[in] Hash of the player's CD Key.

challengeresponse

[in] Result of the GenerateAuth() call

callback

[in] Will be called when the authentication is complete (either successful or not).

instance

[in] Pointer that will be passed to the callback function (for your use)

Section Reference: [Gamespy Persistent Storage SDK](#)

PreAuthenticatePlayerPM

Authenticates a player on the Stats server.

```
void PreAuthenticatePlayerPM(  
    int localid,  
    int profileid,  
    char * challengeresponse,  
    PersAuthCallbackFn callback,  
    void * instance );
```

Routine	Required Header	Distribution
PreAuthenticatePlayerPM	<gpersist.h>	SDKZIP

Parameters

localid

[in] Your game-specific reference number for this player

profileid

[in] The profileid of the player being authenticated.

challengeresponse

[in] Result of the GenerateAuth() call

callback

[in] Will be called when the authentication is complete (either successful or not).

instance

[in] Pointer that will be passed to the callback function (for your use)

Section Reference: [Gamespy Persistent Storage SDK](#)

See Also: [PreAuthenticatePlayerCD](#)

SetPersistData

Sets the entire block of persistent data for a user.

```
void SetPersistData(  
    int localid,  
    int profileid,  
    persisttype_t type,  
    int index,  
    char * data,  
    int len,  
    PersDataSaveCallbackFn callback,  
    void * instance );
```

Routine	Required Header	Distribution
SetPersistData	<gpersist.h>	SDKZIP

Parameters

localid

[in] Your game-specific reference number for this player

profileid

[in] The profileid of the player whose data you are setting.

type

[in] The type of persistent data you are setting. Only rw data is settable.

index

[in] Each profile can have multiple persistent data records associated with them. Usually 0 is used.

data

[in] The persistent data to be saved.

len

[in] The length of the data.

callback

[in] Will be called when the data save is complete.

instance

[in] Pointer that will be passed to the callback function (for your use.)

Remarks

The profileid for whom the data is being set MUST have been authenticated already.

If you are setting key\value delimited data, make sure the "len" parameter includes length of the null terminator.

Section Reference: [Gamespy Persistent Storage SDK](#)

See Also: [GetPersistData](#), [SetPersistDataValues](#)

SetPersistDataValues

If you are saving data in key\value delimited format, you can use this function to only set SOME of the key\value pairs. Only the key value pairs you include in they keyvalues parameter will be updated, the other pairs will stay the same.

```
void SetPersistDataValues(  
    int localid,  
    int profileid,  
    persisttype_t type,  
    int index,  
    gsi_char * keyvalues,  
    PersDataSaveCallbackFn callback,  
    void * instance );
```

Routine	Required Header	Distribution
SetPersistDataValues	<gpersist.h>	SDKZIP

Parameters

localid

[in] Your game-specific reference number for this player

profileid

[in] The profileid of the player whose data you are setting.

type

[in] The type of persistent data you are setting. Only rw data is settable.

index

[in] Each profile can have multiple persistent data records associated with them. Usually 0 is used.

keyvalues

[in] The key/value pairs to be updated.

callback

[in] Will be called when the data save is complete.

instance

[in] Pointer that will be passed to the callback function (for your use.)

Remarks

The profileid for whom the data is being set MUST have been authenticated already.

Section Reference: [Gamespy Persistent Storage SDK](#)

See Also: [GetPersistDataValues](#), [SetPersistData](#)

Persistent Storage SDK Callbacks

[PersAuthCallbackFn](#)

Used in conjunction with the PreAuthenticatePlayer functions; called when the authentication is complete.

[PersDataCallbackFn](#)

Used in conjunction with the GetPersistData functions; called when the data retrieval is complete.

[PersDataSaveCallbackFn](#)

Used in conjunction with the SetPersistData* functions; called when the data save is complete.

[ProfileCallbackFn](#)

Used in conjunction with the GetProfileIDFromCD function; called when the ProfileID retrieval has completed.

PersAuthCallbackFn

Used in conjunction with the PreAuthenticatePlayer functions; called when the authentication is complete.

```
typedef void (*PersAuthCallbackFn)(  
    int localid,  
    int profileid,  
    int authenticated,  
    gsi_char * errmsg,  
    void * instance );
```

Routine	Required Header	Distribution
PersAuthCallbackFn	<gpersist.h>	SDKZIP

Parameters

localid

[in] The localid associated with the authentication.

profileid

[in] The profileid for the player which is being Authenticated.

authenticated

[in] True if authenticated.

errmsg

[in] Error message if applicable.

instance

[in] The same instance as passed into the PreAuthenticate function.

Section Reference: [Gamespy Persistent Storage SDK](#)

PersDataCallbackFn

Used in conjunction with the GetPersistData functions; called when the data retrieval is complete.

```
typedef void (*PersDataCallbackFn)(  
    int localid,  
    int profileid,  
    persisttype_t type,  
    int index,  
    int success,  
    time_t modified,  
    char * data,  
    int len,  
    void * instance );
```

Routine	Required Header	Distribution
PersDataCallbackFn	<gpersist.h>	SDKZIP

Parameters

localid

[in] The localid associated with the data retrieval.

profileid

[in] The profileid associated with the data retrieval.

type

[in] The type of persistent data that was retrieved.

index

[in] The persistent data index, as passed to GetPersistData.

success

[in] True if successful.

modified

[in] Modification time.

data

[in] The data requested.

len

[in] The length of the data buffer

instance

[in] The same instance pointer that was passed to GetPersistData.

Section Reference: [Gamespy Persistent Storage SDK](#)

See Also: [GetPersistData](#), [SetPersistData](#)

PersDataSaveCallbackFn

Used in conjunction with the SetPersistData* functions; called when the data save is complete.

```
typedef void (*PersDataSaveCallbackFn)(  
    int localid,  
    int profileid,  
    persisttype_t type,  
    int index,  
    int success,  
    time_t modified,  
    void *instance );
```

Routine	Required Header	Distribution
PersDataSaveCallbackFn	<gpersist.h>	SDKZIP

Parameters

localid

[in] The localid associated with the save request.

profileid

[in] The profileid associated with the save request.

type

[in] The type of persistent data being saved.

index

[in] The persistent data index, as passed to SetPersistData.

success

[in] True if success

modified

[in] Modification time

instance

[in] The same instance pointer that was passed to SetPersistData.

Section Reference: [Gamespy Persistent Storage SDK](#)

ProfileCallbackFn

Used in conjunction with the GetProfileIDFromCD function; called when the ProfileID retrieval has completed.

```
typedef void (*ProfileCallbackFn)(  
    int localid,  
    int profileid,  
    int success,  
    void * instance );
```

Routine	Required Header	Distribution
ProfileCallbackFn	<gpersist.h>	SDKZIP

Parameters

localid

[in] The localid associated with the ProfileID

profileid

[in] The requested profileid.

success

[in] True if successful.

instance

[in] The same instance pointer that was passed to GetProfileIDFromCD.

Section Reference: [Gamespy Persistent Storage SDK](#)

Persistent Storage SDK Enumerations

[persisttype_t](#)

Type of persistent data stored for each player

persisttype_t

Type of persistent data stored for each player.

```
typedef enum  
{  
    pd_private_ro,  
    pd_private_rw,  
    pd_public_ro,  
    pd_public_rw  
} persisttype_t;
```

Constants

pd_private_ro

Readable only by the authenticated client it belongs to, can only be set on the server.

pd_private_rw

Readable only by the authenticated client it belongs to, set by the authenticated client it belongs to.

pd_public_ro

Readable by any client, can only be set on the server.

pd_public_rw

Readable by any client, set by the authenticated client it belongs to.

Section Reference: [Gamespy Persistent Storage SDK](#)

Stats and Tracking SDK

Overview

The GameSpy Stats and Tracking SDK provides a simple, secure way to report the results and statistics of games to a central server. These results can then be used to help facilitate online rankings, ladders, and tournaments. Tracking is done in a very abstract manner than can be applied to any type of multiplayer game (we provide many examples to demonstrate this, you simply need to find the one that best matches your game) and the results can be displayed and interpreted in a game-specific format.

When used in combination with GameSpy Arcade, the CDKey SDK, or the Presence and Messaging SDK, the Tracking SDK can uniquely and securely identify players and verify their identities for ranking and ladder purposes. Any of the above products provide the unique identification information needed for tracking of individual players.

Game data is sent to the tracking server in the form of "snapshots." These snapshots include information about what has happened in the game, the settings in use, and the players involved. For example, a simple deathmatch game might include information like:

- The map being played
- The players that are playing (including their unique identifying information)
- The score, ping, number of deaths, and other information for each player
- The server settings (such as timelimit, game type, etc)

This data provides a summary of the game that can then be used to update rankings, players statistics, and ladders / tournaments.

Almost any type of data or statistic can be sent using these snapshots; the examples in Appendix A demonstrate just a few of the possibilities. We are more than happy to help you decide what items are important to

track for your game.

You can compose these snapshots manually (the format is described below) or use the bucket system, which is included in the SDK. With the bucket system, each value in the snapshot is assigned a bucket. Buckets can contain integer, real, or string values. For example, each player in the game might have a score bucket that contains the player's score, and each team in the game might have a color bucket that gives that team's color. These buckets support standard operations like addition, subtraction, averaging, concatenation, and more that make them easy to add to your game. Buckets use fast hash-table based lookups and will not affect the performance of your game at all.

Security has been given the utmost consideration in the design of the stats and tracking system, and this documentation includes a full rundown of the security guarantees that can be made (and those that can't). We fully disclose any potential holes or possible areas of exploitation for you to be aware of. Our realistic view of security means that you will never be misled into thinking the system is more secure than it really is, and you will never lose face because you assumed something was secure that wasn't.

We fully support disk based logging in case of a loss of network connectivity or server downtime, however, we don't recommend this for all games because of the security implications it carries.

How It Works (Low-Level Description)

The following terms are used throughout this document.

Server

The machine that is "hosting" the game and to which the clients connect

Host

Same as a server

Client

A single player / machine that connects to a server / host

User

Same as a client

Process:

1. On startup, the host connects to our tracking server, is authenticated, and is assigned a unique connection ID. If disk logging is enabled (see below) and there are logged games, they are sent to the tracking server.
2. When the actual game starts, the host sends a new game notification to the tracking server and creates internal structures for managing the game information.
3. During the game the host collects information into buckets (or developer's own data structures) and sends out snapshots at regular intervals (in case the host is reset before the game finishes)
4. (If player authentication is used) As players connect, the host sends out a challenge to the client, which formats a response based on its password or CD Key. This response is sent back to the host and stored as part of the snapshot.
5. When the game is complete, a final snapshot is sent to the tracking server.
6. A new game can be started immediately over the same connection (multiple simultaneous games over the same tracking server connection are supported as well).

7. The tracking server post-processes the data to extract some standard information and verify the authentication of the players. Disk logged or unusual games are marked for inspection.

The connection to the tracking server is TCP, and all transactions (aside from the initial connection) are non-blocking.

In the event that the tracking server is unavailable, game snapshots will be logged to disk if disk logging is enabled or ignored if disk logging is disabled. Either way your game doesn't have to worry about it, although it can determine whether the connection is working at any time.

There is no actual limit to the size of snapshots, although we recommend you choose what data you send wisely to conserve space. Most games will probably use snapshots of 1KB-8KB (even with 32-64 players).

You can send as many snapshots as you want, but it is recommended that you not send them more than once every 2-3 minutes. Each snapshot replaces the previous one, so sending them more often simply means that in the event of a host crash, more recent data will be preserved. Many games could probably get by with a single, final snapshot, or a midway + final snapshot.

Security Analysis

No Internet game tracking system is perfectly secure. While we believe our security is better than many similar systems, we won't pretend to be perfect. If anyone tells you their system is 100% secure they are either ignorant or lying outright.

There are three levels of security for the system:

- Host run by trusted person (e.g. an official server or server being run by a trusted service)
- Host run by an untrusted person with disk logging disabled (insecure host)
- Host run by an untrusted person with disk logging enabled (insecure host)

If the host is run by a trusted authority, the entire system is extremely secure. Clients cannot authenticate as anyone besides themselves without a valid password. All snapshot information is communicated over the safe link between the trusted host and the tracking server. Disk logging is safe as well since no 3rd parties can access the trusted host's logs. Overall, hosts run by a trusted authority are the ONLY way to guarantee 100% accurate reporting and the ONLY way we recommended running servers for large tournaments and ladders (especially where prizes are involved). Note that even with a trusted server you can never verify the actual identity of the client without physically seeing them. All you can say is that the client playing knows the password of the authenticated client.

Allowing untrusted persons to run a host adds a huge amount of security risk to the system (since the snapshots are generated by the insecure host), and while the vast majority of insecure hosts will report accurately, in the end there is no way to say that any information received from an insecure host is accurate, or that the host is returning information at all. The most simple hack for an insecure host is to not report any stats at all. This in itself may cause problems (e.g. two players playing a ladder match, and the host ends up losing and does not report).

The host authenticates with the tracking server at initial connection using a secret key challenge/response mechanism. All communication between host and tracking server is encrypted with a multi-byte XOR encryption algorithm based on hidden internal keys. Disk based logging is encrypted with a similar system, and an additional checksum algorithm is used to ensure the data has not been tampered with.

Breaking these encryption routines and checksums is possible (since all of the code and keys reside on the host), however it would not be trivial. Most hackers would likely attack from another angle (as described below) to circumvent the checksum and encryption algorithms. However, we will from now on assume that a hacker has broken both the encryption and checksum routines, can read encrypted data as plain text, and can re-encrypt changed data with the correct checksums. With enough incentive, a hacker will extract the encryption and checksum routines, so assuming anything else would be foolish.

Under these assumptions it is clear to see why disk logging is very risky - a hacker can log a real game to disk, then edit this log to reflect different results, and have the game send the edited log to the tracking server. This can all be done without changing one byte of code in the host executable (although the hacker would have to understand the code quite well to figure out the encryption and checksum routines).

Instead of taking the time to decrypt the data, most hackers will likely try to find and change the data before it is encrypted and checksummed. The process of finding the unencrypted data is easier than determining the encryption/checksum algorithms, however changing the data in memory before it is encrypted is very tedious and time consuming, and changing the data significantly requires large byte-code modifications to the host executable (something that is at least as difficult as breaking the encryption/checksum routines).

There are several precautions that have been taken to make finding and changing the data even more difficult, since most amateur hackers will give up after a few hours if they are unable to find a starting point to begin their hack. For example, we obfuscate all (important) strings used in the SDK to make it harder to find the SDK code in the rest of your executable. Our challenges are based on the connection and session id's

to help spot strange values. We use different keys for encrypting different things.

Doing a checksum on your executable and sending it as part of the snapshot can help insure that the host has not been tampered with. Individually these protections can be worked around by a good hacker, however taken together, there is a chance that a hacker will miss one of the protections and be caught as a result. Our tracking server marks all unusual snapshots for later analysis.

One important feature of our system is that even with an insecure host, the host can never steal another user's "identity" and pose as them, or report invalid stats for them on another server. Player authentication is based on a challenge/response system. A challenge (which originates from the tracking server) is sent to the client and a response is generated based on a hash of the challenge and the client's password. This hash eventually makes its way back to the tracking server where it can be verified with the challenge and password that the tracking server knows. The challenge and response in itself are not sufficient to determining the client's password, and the same response cannot be used for more than one connection (since the challenge constantly changes). This means that packet sniffing the client to server connection will not gain any useful information.

If disk based logging is enabled, the challenge will be generated by the host. Because the tracking server is not involved, it cannot guarantee that this challenge will always be different, and thus a host could (after a valid client connects once) reuse the same challenge/response in a different disk log. This is again assuming that the insecure host has figured out the disk logging encryption / checksum algorithms and can change data at will.

So How Secure Is It?

After all of that you may be concerned about how secure the system really is. From our perspective, the system has excellent security in a trusted environment to use for large tournaments and ladders. You can feel confident that the client is either the registered player or has the password of the registered player, which is the best assurance you can

get without physically seeing the person play.

In an unsecure environment we believe the system to still be very secure if it is only used for "fun." By "fun" we mean that the statistics, rankings, and tracking data is not of significant value. Value can be more than just monetary however, so it is important to understand how players will regard the system and its uses. The important thing to remember is that even if a hacker were to break the system, the "value" (both gained by the hacker and potentially lost by the developer) should be minimal and players should understand that "breaking" the system will not gain them anything in the long run. While this will not assure that someone will not break (or attempt to break) the system, it will guarantee that the repercussions of the break will not override the value of the system.

Authentication

The stats and tracking backend can use three different methods to authenticate players. Depending on your game, you can choose which of these methods is best.

Player Name

By default, if no other authentication information is provided, players will be tracked by their name, as reported to the stats system. Obviously this is the lowest level of security, as any player can play as another player by changing their nick name. In some situations this level of security may be sufficient (e.g. when there are no rankings or ladders involved, just game results).

Presence and Messaging Profile ID (pid)

The tracking system supports tracking by an ID and password for the GameSpy Presence and Messaging system. This system (available as a separate SDK) allows players to maintain different profiles with passwords that they can log in with. Even if you aren't using the full presence and messaging SDK, you can still use the account creation / maintenance components to create unique accounts for player tracking and ranking.

CD Key

If your game comes with unique CD Keys you can do tracking via the keys. The CD Keys themselves are not sent on the wire, so you don't have to worry about people setting up servers to steal the keys. Unique identification is based on the combination of the CD Key and the Nickname, so a single CD Key user can have multiple "profiles" by using different nicks.

To use profile ID based authentication, you need to ask the user for their profile ID and password on the client (note: This info can be passed in on the command line for games supported by GameSpy Arcade). The profile ID should be sent to the host during connection and included as the "pid" key for that player. The password is sent to the GenerateAuth function along with the challenge from the host to generate an authentication token, which should be sent back to the host and included as the "auth" key for that player. In the actual snapshot this info will look like (for a

single player):

```
\pid_3\4364342\auth_3\3eaf547cf31de5946aefc3148765d3
```

Using CD Key authentication is similar, except that instead of a profile ID you will send the CD Key hash, which can be obtained using `gcd_getkeyhash` (on the host) in the CD Key SDK. The auth value is based on the un-hashed CD Key and the challenge value and should be generated on the client using the `GenerateAuth` function, as above. In the snapshot this will look like:

```
\cdkey_3\1a353adf3263adfe3298adce37dfac73\auth_3\3ea
```

Note that, by default, all authentication is done post-game. The snapshots are recorded on the tracking server and analyzed after the game is marked as complete. If a player's authentication is incorrect it will be flagged as an error there, and will show up in the game display for that game. If you require pre-game authentication (e.g. a player must be authenticated before they enter the game) we can provide that as a separate service.

Snapshots

All of the stats and tracking is done in the form of "snapshots". Each snapshot gives full information about anything you want to track related to the server, players, or teams. Snapshots are formatted similar to Developer Spec query replies, namely, as key\value backslash delimited pairs. You can track virtually any type of information about the game using snapshots, as the examples in Appendix A demonstrate. If you have a something that you'd like to track with snapshots but aren't sure how, please contact us.

There are three types of keys used in snapshots:

- Server Keys (\keyname\)
- Player Keys (\keyname_N\) where N is the player number, starting at 0
- Team Keys (\keyname_tN\) where N is the team number, starting at 0

Most games that allow players to join and leave during the game reuse existing "slots" and player numbers (e.g. if you have players 0 1 2, and players 1 leaves, the next person to join will be player 1 again).

In order to report accurate stats for ALL players, each player needs to have a unique ID. The Stats SDK Bucket system handles all of this for you. If you create snapshots manually, you will need to make sure that each player has a unique number.

Snapshots describe a "game." What defines a game, or the boundary between two games, is game specific. For example, a "game" may consist of a single level/map. When the level starts, a new "game" is started with a fresh snapshot. When the level ends a final snapshot is sent.

You can send multiple snapshots during the course of the game. Each snapshot will replace the previous one. If the game is still in progress (i.e. more snapshots are possible) send the snapshots with the type of SNAP_UPDATE. Once the game is completed, send a final snapshot

with a type of SNAP_FINAL. This will signal the backend that the game is complete and ready to be processed.

The following is a list of "standard" keys that you may want to use in your snapshots. You are free to add your own keys for game-specific information.

Standard Keys

gamever

followed by a version specifier (x.yy format preferred)

location

followed by a 5 digit numeric string (for a US Zip code) or a 2 letter country code (for Non-US).

hostname

followed by a descriptive host-defined string (can include spaces) that identifies the server (e.g. "Joe's Game!")

mapname

followed by the map name (either filename or descriptive name)

gametype

string which specifies the type of game, or the mod being played.

maxplayers

numeric string, max number of players for this server

fraglimit

number of total kills or points before a level change

timelimit

amount of total time before a level change occurs

player_N

followed by a string which specifies a player name (may include spaces)

score_N

numeric string that contains the score (kills/points) for player N

scoreY_N

numeric string that contains the score (kills/points) for player N against player Y

`skill_N`

a skill rating, if applicable, for player N

`ping_N`

the ping for player N

`team_N`

the team player N is on, either numeric or string

`deaths_N`

number of deaths a player has had

`pid_N`

profileid for the player

`cdkey_N`

hashed CD key of the player

`auth_N`

authentication reply to the given challenge (based on password or cdkey)

`ctime_N`

the "connect time" for this player (in any format, secs recommended)

`dtime_N`

the "disconnect time" for this player (in any format, -1 or empty for still connected)

`ping_N`

ping for this player

`team_tN`

the name for team N

`score_tN`

the score for team N

`ctime_tN`

connect/formation time for team N

`dtime_tN`

disconnect/disbanding time for team N

`serverck`

a checksum for the server executable

gamemode

current mode of the game, e.g. openplaying, exiting, etc.

state

a string that gives a text description of the current state

File Manifest

The following files should be included with this package. If any of the files are missing, please [contact devsupport@gamespy.com](mailto:devsupport@gamespy.com).

File	Description
<i>gstats.c</i>	API code
<i>gstats.h</i>	API header file
<i>gbucket.c,h</i>	Bucket helper code and header
<i>nonport.c,h</i>	System dependent code and header
<i>darray.c,h</i>	Dynamic array code and header
<i>md5c.c,h</i>	MD5 hash code and header
<i>\statstest\</i> Tracking SDK	Example and test code for the Stats and
<i>gstats.dsw</i>	Devstudio project for API and sample code

In addition, to build the SDK and samples, you will need to separately download the GameSpy "common code" package, which includes the shared SDK code used by this SDK and others.

When extracting this package, make sure you preserve the directory tree in order to assure that the code builds correctly.

Implementation

The following is a quick rundown of the basic steps needed to support the SDK. The reference documentation much more extensive documentation for each function.

Step 1: Initialize the Tracking Server Connection

Before the actual gameplay begins you will probably want to connect to the tracking server. You are required to be connected to the tracking server to call [NewGame](#) or [SendGameSnapshot](#) (unless disk based logging is enabled).

First, set the global [gcd_gamename](#) and [gcd_secret_key](#) variables to your gamename and secret key. If these values aren't set correctly, you will be unable to connect to the tracking server.

Once these values are set, call

```
int InitStatsConnection(int gameport)
```

...with the game port that the host is running on. If your game doesn't use multiple ports (or doesn't support more than one host per machine) then you can just use 0.

This call is blocking and make take 1-2 seconds to complete the authentication process. This is the only blocking call in the SDK; all other calls will return immediately.

Step 2: Create A New Game

When the game begins you will want to call

```
statsgame_t NewGame(int usebuckets)
```

If you are going to be using bucket based logging pass 1 for [usebuckets](#), otherwise pass 0.

If you are going to be running more than one game at a time on the host, you will need to store the returned value to pass into the rest of the SDK functions, otherwise you can ignore it and just pass `NULL` (they will use the last game created).

Step 3: Fill In Server Information Buckets (If Using Buckets)

Once you've started the game you'll probably want to fill in some of the basic server information buckets like hostname, mapname, gamever, etc. You should use the `BucketStringOp`, `BucketIntOp`, and `BucketFloatOp` functions to do this. These "op" functions provide the basis for all bucket operations. These functions are actually implemented as macros, but this should not matter for your implementation.

The prototypes are similar, the only difference being the type of the value parameter (String, Int, or Float).

```
Bucket(type)Op(game, name, operation, value, bucketl
```

game

The game to send containing the bucket you want to operate on. If set to `NULL`, the last game created with `NewGame` will be used.

name

The name of the bucket to update.

operation

`bo_set`, `bo_add`, `bo_sub`, `bo_mult`, `bo_div`, `bo_concat`, or `bo_avg`

value

Argument for the operation (bucket OP= value, e.g. bucket += value, bucket *= value)

bucketlevel

`bl_server`, `bl_team`, or `bl_player`. Determines whether you are referring to a server, player, or team bucket

index

For player or team buckets, the game index of the player or team (as passed to `NewPlayer` or `NewTeam`). This will be translated to the actual index internally. Not used for server buckets (`bl_server`).

This will probably look something like:

```
BucketStringOp(game, "hostname", bo_set, hostname->str
BucketStringOp(game, "gamever", bo_set, GAMEVERSION, bl
BucketStringOp(game, "mapname", bo_set, level.mapname,
BucketIntOp(game, "arena", bo_set, arena, bl_server, 0
BucketIntOp(game, "rounds", bo_set, settings.rounds, b
BucketIntOp(game, "round", bo_set, 1, bl_server, 0);
BucketIntOp(game, "armor", bo_set, settings.armor, bl_
BucketIntOp(game, "health", bo_set, settings.health, b
```

Step 4: Create/Remove Players and Teams (If Using Buckets)

As players enter the game and teams are created (if your game has teams) you need to call the [NewPlayer](#) and [NewTeam](#) functions. Calling [NewPlayer](#) or [NewTeam](#) creates a bucket for that player/team's information and allocates that player/team a unique number. It also creates the [ctime](#) bucket for connect time and sets the value.

The prototypes for these functions are:

```
void NewPlayer(statsgame_t game,int pnum, char *name
void NewTeam(statsgame_t game,int tnum, char *name);
```

game

The game to add the player to. If [NULL](#), the last game created with [NewGame](#) will be used.

pnum

Your internal reference for this player, use this value in any calls to the [Bucket___Op](#) functions.

name

The name for this player. If you don't have one yet, set it to empty ("") then call

```
BucketStringOp(game, "player", bo_set, realplayern
```

when you get a real playername.

If players/teams can leave during the game you should call the [RemovePlayer](#) and [RemoveTeam](#) functions so that the disconnect times can be set correctly.

Step 5: Authenticate Clients

If you support authentication via one of the methods described above, you will need to authenticate clients as they connect. To authenticate a client, send them a challenge (as obtained by calling [GetChallenge](#)) and have the client calculate a response (using [GenerateAuth](#)) based on that client's "password".

Note that in the case of CD Key authentication, the "password" is the unencoded (plain text) CD Key. Send this response back to the server along with the identifier (cd key hash or profile id) for the player. Also note that in the case of the CD Key SDK, the server already has the cd key hash for the player (it can be obtained by calling [gcd_getkeyhash](#) in the CD Key SDK). The server then sets these values in the appropriate buckets (or stores in the client structures if not using buckets).

This will look something like:

1. *Server* (send the challenge to the client)

```
challenge = GetChallenge(game);
```

2. *Client* (send the response and profile id to the server)

```
GenerateAuth(challenge, my_password, response);
```

3. *Server*

```
BucketIntOp(game, "pid", bo_set, pid_number, bl_pl  
BucketStringOp(game, "auth", bo_set, response, bl_p.
```

Step 6: Update Buckets As Game Progresses (If Using Buckets)

As the game plays out you will want to update the appropriate buckets as changes occur. For example, when a player scores a point, you will want to update the score bucket for that player. Basically every place in your code where you change data that you want reported, you should make a bucket call.

If you want to refer to other players or teams in your key names, you need to use [GetPlayerIndex](#) and [GetTeamIndex](#) to get the translated (unique) index values for that player/team. For example, to record "Doom-square" style stats of "player A killed player B 10 times", you need to use the a player key in the form of [scoreY_N](#). Y is the translated index of player B (obtained by [GetPlayerIndex](#)). N is the translated index of player A (which is set automatically for player buckets).

Here are some examples of typical modifications to game code needed to add buckets:

Your code:

```
players[i].score += 1;  
players[j].deaths += 1;
```

Addition:

```
BucketIntOp(game, "score",bo_add, 1,bl_player, i);  
BucketIntOp(game, "deaths",bo_add, 1,bl_player, j);  
sprintf(keyname, "score%d",GetPlayerIndex(j));  
BucketIntOp(game, keyname,bo_add, 1,bl_player, i);
```

(note: the above code adds a kill by player i against player j)

Your code:

```
mygame.globals.timelimit = newvalue;
```

Addition:

```
BucketIntOp(game, "timelimit", bo_set, newvalue, bl_se
```

Step 7: Send Snapshots

During the game you can send snapshots so that if the server crashes, at least some information about the game will be preserved. Games which are never "complete" (as signaled by a final snapshot) are flagged as such in the tracking backend, and how they are handled for rankings and ladders is implementation dependant. How many of these "update" snapshots you send is up to you, although sending more than 3 per game is probably overkill. For example, if your game typically runs 30 minutes, then sending a snapshot every 10 minutes would probably be fine. Once the game is complete, send a final snapshot.

The prototype for SendGameSnapShot is:

```
int SendGameSnapShot(statsgame_t game, const char *s
```

game

The game to send a snapshot for. If set to `NULL`, the last game created with `NewGame` will be used.

snapshot

The snapshot to send. If you are using buckets, this will not be used, so you can pass in `NULL`.

final

If this is `SNAP_UPDATE`, the game is marked as in progress, if it is `SNAP_FINAL`, the game is marked as complete.

If you are using buckets, you don't need to worry about building the snapshot - it is built automatically based on the current contents of the buckets. If you are building snapshots yourself, you'll need to make sure that all keys and values are stripped of "\" characters, that players/teams are uniquely identified, and that their numbering is contiguous, starting from zero.

Step 8: Calling the Think function

During the game, if you are connected to the stats server for periods longer than 5 minutes, you'll need to call the function [StatsThink](#). [StatsThink](#) should be called periodically to remove keep alives from the socket buffer. This function should be called at least once every 5 minutes.

Step 9: Send A Final Snapshot And Free The Game

Once the game is complete, send a final snapshot (set [final](#) to [SNAP_FINAL](#) in [SendGameSnapShot](#)) then call [FreeGame](#) to free the game and buckets (if you are using buckets).

Note that this does NOT close the connection to the tracking server. You can create a new game immediately (using [NewGame](#)) or at a later time without re-opening the tracking server connection. The tracking server connection is only closed when you call [CloseStatsConnection](#), which should be done when shutting down, or when no more games are going to be sent.

Appendix: Sample Snapshots

The following are examples of what we think snapshots for some common multiplayer games might look like (including the type of data interesting to record, and how it would be recorded). You are free to add your own game-specific data as needed to the snapshot.

The other purpose of creating these examples was to prove that the snapshot system can record ANY type of interesting game data (in lieu of a time-based logging format).

You can view these samples using the "validator" application including in the SDK (just cut/paste). In addition to the standard keys from the list above, each game has some game-specific keys with their descriptions.

18 Player Rocket Arena 2 Pickup (Team) Match

Key Descriptions:

arena

the arena number this match is taking place in

armor

initial armor value

armorprotect

armor protection setting

compmode

whether competition mode is on

damagescoring

whether damage scoring is on

fallingdamage

whether falling damage is on

health

initial health value

healthprotect

health protection setting

round

the current round number for the match

rounds

the max number of rounds

railkills_N

number of kills with the rail by this player

rocketkills_N

number of kills with the RL by this player

grenadekills_N

number of kills with the grenade launcher by this player

otherkills_N

number of kills with other weapons by this player

suicides_N

number of self-kills by this player

Completed:

```
\round\8\suicides_0\0\rocketkills_11\3\score_14\33\s
```

1v1 Quake 3 Competition Match

Key Descriptions:

gameck

a checksum for the game dll

rocketK_N

number of kills with the RL by this player

railK_N

number of kills with the Railgun by this player

shotgunK_N

number of kills with the Railgun by this player

selfK_N

number of suicides

`armorPU_N`

total amount of armor picked up by this person

`rocketsPU_N`

total number of rockets picked up

`slugsPU_N`

total number of slugs picked up

`rockethitp_N`

hit percentage for rocket shots

`railhitp_N`

hit percentage for rail shots

In Progress:

```
\hostname\My_1v1_Server\mapname\q3map23\timelimit\20
```

Completed:

```
\hostname\My_1v1_Server\mapname\q3map23\timelimit\20
```

Quake 3 FFA Server

Key Descriptions:

`gameck`

a checksum for the game dll

`rocketK_N`

number of kills with the RL by this player

`railK_N`

number of kills with the Railgun by this player

`shotgunK_N`

number of kills with the Railgun by this player

`selfK_N`

number of suicides

armorPU_N

total ammount of armor picked up by this person

rocketsPU_N

total number of rockets picked up

slugsPU_N

total number of slugs picked up

rockethitp_N

hit percentage for rocket shots

railhitp_N

hit percentage for rail shots

Completed

```
\hostname\Random_Deathmatch_Server\mapname\q3map20\t
```

4v4 Team Quake 3 Competition

Key Descriptions:

gameck

a checksum for the game dll

rocketK_N

number of kills with the RL by this player

railK_N

number of kills with the Railgun by this player

shotgunk_N

number of kills with the Railgun by this player

selfK_N

number of suicides

armorPU_N

total ammount of armor picked up by this person

rocketsPU_N

total number of rockets picked up

`slugsPU_N`

total number of slugs picked up

`rockethitp_N`

hit percentage for rocket shots

`railhitp_N`

hit percentage for rail shots

`tkills_N`

Number of teammates a player has killed

`quads_tN`

Number of Quad powerups team N has gotten

`avglifespan_tN`

Average lifespan for a player in team N

In progress:

```
\hostname\TeamPlay_Competition_Server\mapname\q3map2
```

Completed:

```
\hostname\TeamPlay_Competition_Server\mapname\q3map2
```

1v1 Starcraft Match

Key Descriptions:

`elapsedtime`

total time this match has taken

`startpos`

relative starting position on the map

`gasmined`

total gas resources mined

`mineralsmined`

total mineral resources mined

tspent

total resources spent

unitsP

total units produced

unitsK

number of enemy units killed

unitsL

number of own units lost

structuresP

total number of structures erected

structuresK

number of enemy structures razed

structuresL

number of own structures lost

race

which race the player played

Completed:

```
\hostname\crt_vs_walla\mapname\Lost Temple\gametype\  
\state\Game Complete, crt wins\elapsedtime\36:49\pla  
\startpos_0\Center Right\gasmined_0\10410\mineralsmi  
\unitsP_0\453\unitsK_0\140\unitsL_0\254\structuresP_  
\structuresL_0\1\race_0\Zerg\pid_0\23432\auth_0\cc7a  
\Tspent_1\32438\unitsP_1\183\unitsK_1\254\unitsL_1\1
```

1v1v1v1 Starcraft Match

Key Descriptions:

elapsedtime

total time this match has taken

startpos

relative starting position on the map

gasmined

total gas resources mined

mineralsmined

total mineral resources mined

tspent

total resources spent

unitsP

total units produced

unitsK

number of enemy units killed

unitsL

number of own units lost

structuresP

total number of structures erected

structuresK

number of enemy structures razed

structuresL

number of own structures lost

race

which race the player played

In progress:

```
\hostname\4 Player Game!\mapname\Hellhole\gametype\M
```

Completed:

```
\hostname\4 Player Game!\mapname\Hellhole\gametype\M
```

2v2 Starcraft Match

Key Descriptions:

elapsedtime
total time this match has taken

startpos
relative starting position on the map

gasmined
total gas resources mined

mineralsmined
total mineral resources mined

tspent
total resources spent

unitsP
total units produced

unitsK
number of enemy units killed

unitsL
number of own units lost

structuresP
total number of structures erected

structuresK
number of enemy structures razed

structuresL
number of own structures lost

race
which race the player played

Completed:

```
\hostname\2v2 Teams\mapname\Hellhole\gametype\Team M  
\unitsL_1\152\structuresP_1\14\structuresK_1\7\struc  
\unitsL_2\145\structuresP_2\13\structuresK_2\6\struc  
\unitsL_3\132\structuresP_3\16\structuresK_3\0\struc
```

2 Human vs. 1 Computer Starcraft Match

Key Descriptions:

`elapsedtime`

total time this match has taken

`startpos`

relative starting position on the map

`gasmined`

total gas resources mined

`mineralsmined`

total mineral resources mined

`tspent`

total resources spent

`unitsP`

total units produced

`unitsK`

number of enemy units killed

`unitsL`

number of own units lost

`structuresP`

total number of structures erected

`structuresK`

number of enemy structures razed

`structuresL`

number of own structures lost

`race`

which race the player played

`aiplayer_N`

whether or not the player is controlled by the computer

Completed:

```
\hostname\2 vs Computer\mapname\Sherwood Forest\game
```

4v4 Rainbow 6 Team DM

Key Description:

`sex_N`

sex (M or F)

`totaltime_N`

total number of seconds for the match

`specialty_N`

specialty (Assault, Recon, Sniper, etc.)

`roundsfired_N`

total rounds fired

`hitpercent_N`

total hit %

`tkills_N`

number of team player's killed

Completed:

```
\hostname\Jim's Server\mapname\Killing Field\gametyp
```

4 vs. Computer Rainbow 6 Coop

Key Description:

`sex_N`

sex (M or F)

`totaltime_N`

total number of seconds for the match

`specialty_N`

specialty (Assault, Recon, Sniper, etc.)

`roundsfired_N`

total rounds fired

`hitpercent_N`

total hit %

tkills_N

number of team player's killed

difficuly

mission's difficulty level (Rookie, Veteran, Elite)

human_N

1 for a human, 0 for an AI player

Completed:

```
\hostname\Jim's Server\mapname\M78 Rescue Gilligan\c
```

1v1 FIFA Soccer

Key Descriptions:

time

total match time in seconds

precipitation

rain, snow, sleet, etc.

surface

grass, turf, etc.

human_N

1 for a human, 0 for an AI player

goals_N

number of goals scored by this player

assists_N

number of assists by this player

steals_N

number of steals by this player

shots_N

number of shots on goal by this player

fouls_tN

number of times this team fouled

`yellowcards_tN`

number of yellowcards received by this team

`redcards_tN`

number of redcards received by this team

Completed:

```
\hostname\John's Cup\mapname\Brazil\gametype\Standar
```

2v2 NBA Live (with 3 computer players on each team)

Key Descriptions:

`time`

length of game in seconds

`points`

number of points scored

`shotpercent_N`

player's shot percentage

`threes_N`

number of 3-pointers

`rebounds_N`

number of rebounds

`blocks_N`

number of blocks

`steals_N`

number of steals

`fouls_N`

number of fouls

`shotpercent_tN`

team's shot percentage

`threes_tN`

number of 3-pointers

`rebounds_tN`

number of rebounds

`blocks_tN`

number of blocks

`steals_tN`

number of steals

`fouls_tN`

number of fouls

`possessionTime_tN`

total time of possession for the team

Completed:

```
\hostname\Play Basketball Here!\mapname\New Jersey\g
```

1 player PGA Tour 99

Key Descriptions:

`nholes`

number of holes on the course

`par`

par for the course

`holeXX_N`

score for player N on hole XX

`score_N`

current score for player N (vs. par)

In progress:

```
\hostname\Single_Player_Game_(Bob)\nholes\9\mapname\
```

Completed:

```
\hostname\Single_Player_Game_(Bob)\nholes\9\mapname\
```

1v1 Checkers

Key Descriptions:

`moves`

total number of moves made in the game

`winner`

player number of the winner

`time_N`

total number of seconds player took to make his moves

`pieceslost_N`

number of pieces this player lost

`kings_N`

number of times this player "kinged" a piece

Completed:

```
\hostname\I play checkers real good.\gametype\Standards
```

3 Player You Don't Know Jack

Key Descriptions:

`correct_N`

number of correct answers

`wrong_N`

number of wrong answers

Completed:

```
\gametype\Toilet Humor\player_0\Snap\score_0\800\cor
```

Stats and Tracking SDK Functions

BucketFloatOp	Performs an operation on a bucket for a game.
BucketIntOp	Performs an operation on a bucket for a game.
BucketStringOp	Performs an operation on a bucket for a game.
CloseStatsConnection	Closes the connection to the stats server. You should do this when done with the connection.
FreeGame	Frees a game and its associated structures (including buckets).
GenerateAuth	Used on the CLIENT SIDE to generate an authentication reply (auth_N) for a given challenge and password (CD Key or Profile password)
GetChallenge	Get the challenge value that should be sent to clients for authentication (using GenerateAuth).
GetPlayerIndex	Gets the gstats reference number for a player.
GetTeamIndex	Gets the gstats reference number for a team.
InitStatsAsync	Initializes the Tracking Server Connection Without Blocking.

InitStatsConnection	Initializes the Tracking Server Connection.
InitStatsThink	Continues InitStatsAsync connection attempt.
IsStatsConnected	Returns whether or not you are currently connected to the stats server.
NewGame	Creates a new game for logging and registers it with the stats server.
NewPlayer	Adds a "player" to the game and assigns them an internal player number. Sets their connect time to the number of seconds since the NewGame function was called.
NewTeam	Adds a "team" to the game and assigns it an internal team number. Sets its connect time to the number of seconds since the NewGame function was called.
RemovePlayer	Removes a "player" from the game and sets their disconnect time to the number of seconds since NewGame was called.
RemoveTeam	Removes a "team" from the game and sets its disconnect time to the number of seconds since NewGame was called.
SendGameSnapShot	Sends a snapshot of information about the current game.
StatsThink	

Called to keep connection open

BucketFloatOp

Performs an operation on a bucket for a game.

```
double BucketFloatOp(  
    statsgame_t game,  
    char * name,  
    bucketop_t operation,  
    double value,  
    bucketlevel_t bucketlevel,  
    int index );
```

Routine	Required Header	Distribution
BucketFloatOp	<gstats.h>	SDKZIP

Return Value

Returns the resultant bucket value.

Parameters

game

[in] The game to send containing the bucket you want to operate on. If NULL, uses most recently created game.

name

[in] The name of the bucket to update.

operation

[in] Indicates the operation to perform on the bucket.

value

[in] Argument for the operation.

bucketlevel

[in] Determines whether you are referring to a server, player, or team bucket.

index

[in] For player or team buckets, the internal reference, as passed to NewPlayer or NewTeam.

Remarks

if the bucket doesn't exist already, the call will set the bucket to whatever "value" is. You can create each bucket explicitly by using the `bo_set` operation with whatever initial value you want the bucket to have.

Each bucket type (int, float, or string) has its own operation function, always call the same one for each bucket (i.e. don't create a bucket with `BucketIntOp` then try to add a float with **BucketFloatOp**).

Section Reference: [Gamespy Stats and Tracking SDK](#)

See Also: [BucketIntOp](#), [BucketStringOp](#)

BucketIntOp

Performs an operation on a bucket for a game.

```
int BucketIntOp(  
    statsgame_t game,  
    char * name,  
    bucketop_t operation,  
    int value,  
    bucketlevel_t bucketlevel,  
    int index );
```

Routine	Required Header	Distribution
BucketIntOp	<gstats.h>	SDKZIP

Return Value

Returns the resultant bucket value.

Parameters

game

[in] The game to send containing the bucket you want to operate on. If NULL, uses most recently created game.

name

[in] The name of the bucket to update.

operation

[in] Indicates the operation to perform on the bucket.

value

[in] Argument for the operation.

bucketlevel

[in] Determines whether you are referring to a server, player, or team bucket.

index

[in] For player or team buckets, the internal reference, as passed to NewPlayer or NewTeam.

Remarks

Performs an operation on a bucket for a game; if the bucket doesn't exist already, the call will set the bucket to whatever "value" is. You can create each bucket explicitly by using the bo_set operation with whatever initial value you want the bucket to have.

Each bucket type (int, float, or string) has its own operation function, always call the same one for each bucket (i.e. don't create a bucket with **BucketIntOp** then try to add a float with BucketFloatOp).

Section Reference: [Gamespy Stats and Tracking SDK](#)

See Also: [BucketFloatOp](#), [BucketStringOp](#)

BucketStringOp

Performs an operation on a bucket for a game.

```
void BucketStringOp(  
    statsgame_t game,  
    char * name,  
    bucketop_t operation,  
    string value,  
    bucketlevel_t bucketlevel,  
    int index );
```

Routine	Required Header	Distribution
BucketStringOp	<gstats.h>	SDKZIP

Parameters

game

[in] The game to send containing the bucket you want to operate on. If NULL, uses most recently created game.

name

[in] The name of the bucket to update.

operation

[in] Indicates the operation to perform on the bucket.

value

[in] Argument for the operation.

bucketlevel

[in] Determines whether you are referring to a server, player, or team bucket.

index

[in] For player or team buckets, the internal reference, as passed to NewPlayer or NewTeam.

Remarks

Performs an operation on a bucket for a game; if the bucket doesn't exist already, the call will set the bucket to whatever "value" is. You can create each bucket explicitly by using the `bo_set` operation with whatever initial value you want the bucket to have.

Each bucket type (int, float, or string) has its own operation function, always call the same one for each bucket (i.e. don't create a bucket with `BucketIntOp` then try to add a float with `BucketFloatOp`).

Example

To set the name of a player whose name was not known when NewPlayer(

```
BucketStringOp(game, "player", bo_set, realplayername, bl_player
```

Section Reference: [Gamespy Stats and Tracking SDK](#)

See Also: [BucketFloatOp](#), [BucketIntOp](#)

CloseStatsConnection

Closes the connection to the stats server. You should do this when done with the connection.

void CloseStatsConnection();

Routine	Required Header	Distribution
CloseStatsConnection	<gstats.h>	SDKZIP

Section Reference: [Gamespy Stats and Tracking SDK](#)

FreeGame

Frees a game and its associated structures (including buckets).

```
void FreeGame(  
    statsgame_t game );
```

Routine	Required Header	Distribution
FreeGame	<gstats.h>	SDKZIP

Parameters

game

[in] The game you want to free. If NULL, uses most recently created game.

Remarks

You should send a final snapshot for the game (using `SendGameSnapShot` with `SNAP_FINAL`) before freeing the game.

Section Reference: [Gamespy Stats and Tracking SDK](#)

See Also: [SendGameSnapShot](#)

GenerateAuth

Used on the CLIENT SIDE to generate an authentication reply (auth_N) for a given challenge and password (CD Key or Profile password).

```
char * GenerateAuth(  
    char * challenge,  
    gsi_char * password,  
    char response[33] );
```

Routine	Required Header	Distribution
GenerateAuth	<gstats.h>	SDKZIP

Return Value

A pointer to the response parameter.

Parameters

challenge

[in] The challenge string sent by the server. On the server this should be generated with `GetChallenge`.

password

[in] The CD Key (un-hashed) or profile password.

response

[out] The output authentication string.

Section Reference: [Gamespy Stats and Tracking SDK](#)

See Also: [GetChallenge](#)

GetChallenge

Get the challenge value that should be sent to clients for authentication (using GenerateAuth).

```
char * GetChallenge(  
    statsgame_t game );
```

Routine	Required Header	Distribution
GetChallenge	<gstats.h>	SDKZIP

Return Value

Returns a string to pass to `GenerateAuth` to create the authentication hash.

Parameters

game

[in] The game to return the challenge string for. If NULL, uses most recently created game.

Remarks

You do not have to free the string when done. This string will be constant for the entire length of the game and is generated during the call to NewGame.

Section Reference: [Gamespy Stats and Tracking SDK](#)

GetPlayerIndex

Gets the gstats reference number for a player.

```
int GetPlayerIndex(  
    statsgame_t game,  
    int pnum );
```

Routine	Required Header	Distribution
GetPlayerIndex	<gstats.h>	SDKZIP

Return Value

Returns the requested index.

Parameters

game

[in] The game for which to retrieve the translated value. If NULL, uses most recently created game.

pnum

[in] Your internal player number, as sent to NewPlayer.

Remarks

As players join and leave, their assigned indexes change. Normally this doesn't matter to you, but if you want to do a key name or key value that references a player or team number, such as setting a player's team number, you need to use the translated values.

Section Reference: [Gamespy Stats and Tracking SDK](#)

See Also: [NewTeam](#), [NewPlayer](#)

GetTeamIndex

Gets the gstats reference number for a team.

```
int GetTeamIndex(  
    statsgame_t game,  
    int tnum );
```

Routine	Required Header	Distribution
GetTeamIndex	<gstats.h>	SDKZIP

Return Value

Returns the requested index.

Parameters

game

[in] The game for which to retrieve the translated value. If NULL, uses most recently created game.

tnum

[in] Your internal team number, as sent to NewTeam.

Remarks

As teams are added and removed, their assigned indexes may not be trackable. Normally this doesn't matter to you, but if you want to do a key name or key value that references a player or team number, such as setting a player's team number, you need to use the translated values.

Section Reference: [Gamespy Stats and Tracking SDK](#)

InitStatsAsync

Initializes the Tracking Server Connection Without Blocking.

```
int InitStatsAsync(  
    int gameport,  
    gsi_time timeout );
```

Routine	Required Header	Distribution
InitStatsAsync	<gstats.h>	SDKZIP

Return Value

GE_NOERROR if successful; otherwise one of the GE_ error codes. See Remarks.

Parameters

gameport

[in] The game port that the host is running on.

timeout

[in] Optional timeout for the connection attempt. (See Remarks)

Remarks

InitStatsThink will need to be called while the connection attempt is in progress.

Be sure to set the global "gcd_gamename" and "gcd_secret_key" variables to your gamename and secret key before making this call or you will be unable to connect to the tracking server.

If your game doesn't use multiple ports (or doesn't support more than one host per machine) then you can just use 0 for the gameport.

A optional timeout value may be supplied to control how long the SDK will attempt to connect. The connection attempt will abort after the elapsed time. The connection attempt may be aborted at any time by calling CloseStatsConnection.

Error return values include:

GE_NODNS: Unable to resolve stats server DNS

GE_NOSOCKET: Unable to create data socket

GE_NOCONNECT: Unable to connect to stats server

GE_DATAERROR: Unable to receive challenge from stats server, or bad challenge

GE_NOERROR: Connected to stats server and ready to send data

GE_CONNECTING: Connect did not immediately complete. Call InitStatsThink to continue.

GE_TIMEOUT: Connect did not complete before timeout value was reached.

Section Reference: [Gamespy Stats and Tracking SDK](#)

See Also: [InitStatsConnection](#), [InitStatsThink](#)

InitStatsConnection

Initializes the Tracking Server Connection.

```
int InitStatsConnection(  
    int gameport );
```

Routine	Required Header	Distribution
InitStatsConnection	<gstats.h>	SDKZIP

Return Value

GE_NOERROR if successful; otherwise one of the GE_ error codes. See Remarks.

Parameters

gameport

[in] The game port that the host is running on.

Remarks

Be sure to set the global "gcd_gamename" and "gcd_secret_key" variables to your gamename and secret key before making this call or you will be unable to connect to the tracking server.

If your game doesn't use multiple ports (or doesn't support more than one host per machine) then you can just use 0 for the gameport.

Error return values include:

GE_NODNS: Unable to resolve stats server DNS

GE_NOSOCKET: Unable to create data socket

GE_NOCONNECT: Unable to connect to stats server

GE_DATAERROR: Unable to receive challenge from stats server, or bad challenge

GE_NOERROR: Connected to stats server and ready to send data.

Section Reference: [Gamespy Stats and Tracking SDK](#)

See Also: [InitStatsAsync](#), [CloseStatsConnection](#)

InitStatsThink

Continues InitStatsAsync connection attempt.

int InitStatsThink();

Routine	Required Header	Distribution
InitStatsThink	<gstats.h>	SDKZIP

Return Value

GE_NOERROR if successful; otherwise one of the GE_ error codes. See Remarks.

Remarks

This function should continue to be called as long as GE_CONNECTING is returned.

Error return values include:

GE_NODNS: Unable to resolve stats server DNS

GE_NOCKET: Unable to create data socket

GE_NOCONNECT: Unable to connect to stats server

GE_DATAERROR: Unable to receive challenge from stats server, or bad challenge

GE_NOERROR: Connected to stats server and ready to send data

GE_CONNECTING: Connect did not immediately complete. Call

InitStatsThink to continue.

GE_TIMEOUT: Connect did not complete before timeout value was reached.

Section Reference: [Gamespy Stats and Tracking SDK](#)

See Also: [InitStatsAsync](#), [CloseStatsConnection](#)

IsStatsConnected

Returns whether or not you are currently connected to the stats server.

int IsStatsConnected();

Routine	Required Header	Distribution
IsStatsConnected	<gstats.h>	SDKZIP

Return Value

Returns 1 if connected, 0 otherwise.

Remarks

Even if your initial connection was successful, you may lose connection later and want to try to reconnect.

If a callback returns unsuccessfully, check this function to see if it was because of a disconnection.

Section Reference: [Gamespy Stats and Tracking SDK](#)

NewGame

Creates a new game for logging and registers it with the stats server.

```
statsgame_t NewGame(  
    int usebuckets );
```

Routine	Required Header	Distribution
NewGame	<gstats.h>	SDKZIP

Return Value

Returns a pointer to the new game.

Parameters

usebuckets

[in] If using bucket based logging, pass 1, otherwise 0.

Remarks

If you are going to be running more than one game at a time on the host, you will need to store the returned value to pass into the rest of the SDK functions, otherwise you can ignore it and just pass NULL (they will use the last game created).

Section Reference: [Gamespy Stats and Tracking SDK](#)

NewPlayer

Adds a "player" to the game and assigns them an internal player number. Sets their connect time to the number of seconds since the NewGame function was called.

```
void NewPlayer(  
    statsgame_t game,  
    int pnum,  
    gsi_char * name );
```

Routine	Required Header	Distribution
NewPlayer	<gstats.h>	SDKZIP

Parameters

game

[in] The game to add the player to. If NULL, uses most recently created game

pnum

[in] Your internal reference for this player.

name

[in] The name for this player.

Remarks

If you don't have the player's name yet, set it to empty ("") and use of the BucketStringOp function to set it later.

Use parameter pnum's internal reference value in any calls to the "Bucket__Op" functions.

Section Reference: [Gamespy Stats and Tracking SDK](#)

NewTeam

Adds a "team" to the game and assigns it an internal team number. Sets its connect time to the number of seconds since the NewGame function was called.

```
void NewTeam(  
    statsgame_t game,  
    int tnum,  
    gsi_char * name );
```

Routine	Required Header	Distribution
NewTeam	<gstats.h>	SDKZIP

Parameters

game

[in] The game to add the team to. If NULL, uses most recently created game

tnum

[in] Your internal reference for this team.

name

[in] The name for this team.

Remarks

If you don't have the team's name yet, set it to empty ("") and use of the BucketStringOp function to set it later.

Use parameter tnum's internal reference value in any calls to the "Bucket__Op" functions.

Section Reference: [Gamespy Stats and Tracking SDK](#)

RemovePlayer

Removes a "player" from the game and sets their disconnect time to the number of seconds since NewGame was called.

```
void RemovePlayer(  
    statsgame_t game,  
    int pnum );
```

Routine	Required Header	Distribution
RemovePlayer	<gstats.h>	SDKZIP

Parameters

game

[in] The game to add the player to. If NULL, uses most recently created game.

pnum

[in] Your internal reference for this player.

Section Reference: [Gamespy Stats and Tracking SDK](#)

RemoveTeam

Removes a "team" from the game and sets its disconnect time to the number of seconds since NewGame was called.

```
void RemoveTeam(  
    statsgame_t game,  
    int tnum );
```

Routine	Required Header	Distribution
RemoveTeam	<gstats.h>	SDKZIP

Parameters

game

[in] The game to add the team to. If NULL, uses most recently created game.

tnum

[in] Your internal reference for this team.

Section Reference: [Gamespy Stats and Tracking SDK](#)

SendGameSnapShot

Sends a snapshot of information about the current game.

```
int SendGameSnapShot(  
    statsgame_t game,  
    const gsi_char * snapshot,  
    int final );
```

Routine	Required Header	Distribution
SendGameSnapShot	<gstats.h>	SDKZIP

Return Value

Returns GE_NOERROR if successful; otherwise, one of the other GE_ error values. See Remarks.

Parameters

game

[in] The game to send a snapshot for. If NULL, uses most recently created game.

snapshot

[in] The snapshot to send. If you are using buckets, you can pass in NULL.

final

[in] SNAP_UPDATE if the game is in progress; SNAP_FINAL if this is the final snapshot.

Remarks

If bucket based logging is enabled the snapshot will be generated from the buckets; otherwise, you should provide it in "snapshot".

Return values include:

GE_NOERROR: The update was sent, or disk logging is enabled and the game was logged.

GE_DATAERROR: If game is NULL and the last game created by NewGame failed

GE_NOCONNECT: If the connection is lost and disk logging is disabled.

Section Reference: [Gamespy Stats and Tracking SDK](#)

Stats and Tracking SDK Enumerations

[bucketlevel_t](#)

The types of buckets (server info, team info, or player info)

[bucketop_t](#)

All of the operations that can performed on a bucket

bucketlevel_t

The types of buckets (server info, team info, or player info).

```
typedef enum  
{  
    bl_server,  
    bl_team,  
    bl_player  
} bucketlevel_t;
```

Constants

bl_server

Bucket contains server info.

bl_team

Bucket contains team info.

bl_player

Bucket contains player info.

Section Reference: [Gamespy Stats and Tracking SDK](#)

See Also: [BucketFloatOp](#), [BucketIntOp](#), [BucketStringOp](#)

bucketop_t

All of the operations that can performed on a bucket.

```
typedef enum  
{  
    bo_set,  
    bo_add,  
    bo_sub,  
    bo_mult,  
    bo_div,  
    bo_concat,  
    bo_avg  
} bucketop_t;
```

Constants

bo_set

Sets the bucket to given value.

bo_add

Adds a value to the bucket.

bo_sub

Subtracts a value from the bucket.

bo_mult

Multiplies the bucket by a value.

bo_div

Divides the bucket by a value.

bo_concat

Concatenates a value to the bucket.

bo_avg

Averages-in a value to the bucket.

Section Reference: [Gamespy Stats and Tracking SDK](#)

See Also: [BucketFloatOp](#), [BucketIntOp](#), [BucketStringOp](#)

adReset

Return the SDK back to its initialization point.

```
AdResult adReset(  
    AdInterfacePtr * theInterface );
```

Routine	Required Header	Distribution
adReset	<ad.h>	SDKZIP

Return Value

This function returns `AdResult_NO_ERROR` upon success. Otherwise a valid `AdResult` error condition is returned. (see remarks)

Parameters

theInterface

[in] AdInterfacePtr to be reset

Remarks

This function will return the SDK back to its starting point. The internal position and ad info lists will be cleared. All usage data will be cleared.

AdResult_INVALID_PARAMETERS will be returned if *theInterface* is invalid.

Example

```
AdInterfacePtr anInterface = NULL;
AdResult       aResult     = AdResult_NO_ERROR;

    // Set run-time parameters
    AdInitParams anInitParams;
    memset(&anInitParams;, 0, sizeof(AdInitParams));
    anInitParams.mGameId = GAME_ID;
anInitParams.mOfflineFilePath = "ad";

    printf("Initializing the Ad SDK\r\n");
    aResult = adInitialize(&anInitParams;, &anInterface;
    if (aResult != AdResult_NO_ERROR)
    {
        printf("adInitialize failed (%d)\r\n", aResu
        return 0;
    }
```

Section Reference: [Gamespy Advertising SDK](#)

See Also: [adInitialize](#)

GPFindPlayerMatch

An element of GPFindPlayersResponseArg, which is the arg parameter passed to a callback generated by a call to gpFindPlayers .

```
typedef struct  
{  
    GPProfile profile;  
    gsi_char nick[GP_NICK_LEN];  
    GPEnum status;  
    gsi_char statusString[GP_STATUS_STRING_LEN];  
} GPFindPlayerMatch;
```

Members

profile

The profile object for this match.

nick

The nick for this match.

status

Status of the match.

statusString

Readable text string representation of the status.

Section Reference: [Gamespy Presence SDK](#)

See Also: [GPFindPlayersResponseArg](#), [gpFindPlayers](#)

GPFindPlayersResponseArg

The arg parameter passed to a callback generated by a call to gpFindPlayers is of this type.

```
typedef struct  
{  
    GPRResult result;  
    int productID;  
    int numMatches;  
    GPFindPlayerMatch * matches;  
} GPFindPlayersResponseArg;
```

Members

result

The result of the find players operation; GP_NO_ERROR if successful.

productID

This is the same product ID that was passed as a parameter to gpFindPlayers.

numMatches

The number of matches found (stored in the matches field).

matches

The search matches.

Section Reference: [Gamespy Presence SDK](#)

See Also: [GPFindPlayerMatch](#), [gpFindPlayers](#)

GPRRecvBuddyAuthArg

Information sent to the GP_RECV_BUDDY_AUTH callback.

```
typedef struct  
{  
    GPProfile profile;  
    unsigned int date;  
} GPRRecvBuddyAuthArg;
```

Members

profile

The profile who authorized the request.

date

The date when the auth was accepted.

Section Reference: [Gamespy Presence SDK](#)

See Also: [gpSetCallback](#), [GPEnum](#), [gpAuthBuddyRequest](#)

gt2CreateAdHocSocket

Creates a new socket, which can be used for making outgoing connections or accepting incoming connections. See `gt2CreateSocket` for details.

gt2CreateAdHocSocket();

Routine	Required Header	Distribution
gt2CreateAdHocSocket	<gt2.h>	SDKZIP

Remarks

AdHoc Sockets use MAC address instead of IP address. See.

Section Reference: [Gamespy Transport SDK](#)

See Also: [gt2CreateSocket](#)

gt2GetLastSentMessageID

Gets the message id for the last reliably sent message. Unreliable messages do not have an id.

```
GT2MessageID gt2GetLastSentMessageID(  
    GT2Connection connection );
```

Routine	Required Header	Distribution
gt2GetLastSentMessageID	<gt2.h>	SDKZIP

Return Value

The message ID of the last reliably sent message.

Parameters

connection

[in] The handle to the connection.

Remarks

This should be called immediately after `gt2Send`. Waiting until after a call to `gt2Think` can result in an invalid message id being returned. Note that the use of filters that can either drop or delay messages can complicate the process, because in those cases a call to `gt2Send` does not guarantee that a message will actually be sent. In those cases, **`gt2GetLastSentMessageID`** should be called after `gt2FilteredSend`, because the actual message will be sent from within that function.

Section Reference: [Gamespy Transport SDK](#)

See Also: [gt2WasMessageIDConfirmed](#)

gt2WasMessageIDConfirmed

Checks if confirmation has been received that the remote end received a particular reliable message.

```
GT2Bool gt2WasMessageIDConfirmed(  
    GT2Connection connection,  
    GT2MessageID messageID );
```

Routine	Required Header	Distribution
gt2WasMessageIDConfirmed	<gt2.h>	SDKZIP

Return Value

GT2True if confirmation was received locally that the reliable message represented by messageID was received by the remote end of the connection, GT2False if confirmation was not yet received.

Parameters

connection

[in] The handle to the connection.

messageID

[in] The ID of the message to check for confirmation.

Remarks

This should only be called on message ids that were returned by `gt2GetLastSendMessageID`, and should be used relatively soon after the message was sent, due to message ids wrapping around after a period of time.

Section Reference: [Gamespy Transport SDK](#)

See Also: [gt2GetLastSendMessageID](#)

gti2IpToMac

Converts a 32 bit IP address to a 48 bit Mac address.

gti2IpToMac();

Routine	Required Header	Distribution
gti2IpToMac	<gt2.h>	SDKZIP

Remarks

Internally every time **gti2IpToMac** is called, the full 48 bit mac address is stored in a look up table. This mac address can be later retrieved using `gti2MacToIp`.

Section Reference: [Gamespy Transport SDK](#)

See Also: [gti2MacToIp](#)

gti2MacToIp

Change mac ethernet to IP address.

gti2MacToIp();

Routine	Required Header	Distribution
gti2MacToIp	<gt2.h>	SDKZIP

Remarks

Internally every time `gti2IpToMac` is called, the full 48 bit mac address is stored in a look up table. This function will find the corresponding entry to the given ip and return the full 48 bit mac address.

Section Reference: [Gamespy Transport SDK](#)

See Also: [gti2IpToMac](#)

StatsThink

Called to keep connection open.

int StatsThink();

Routine	Required Header	Distribution
StatsThink	<gstats.h>	SDKZIP

Return Value

returns 1 for success, or 0 for any errors that occurred.

Remarks

This function is used to consume keep-alives that the server sends to maintain the connection. It does not have to be called very often. Calling this function about once every 10 seconds should be sufficient.

Section Reference: [Gamespy Stats and Tracking SDK](#)

gpFindPlayers

This function finds players who are invitable to the given game.

```
GPRresult gpFindPlayers(  
    GPConnection * connection,  
    int productID,  
    GPEnum blocking,  
    GPcallback callback,  
    void * param );
```

Routine	Required Header	Distribution
gpFindPlayers	<gp.h>	SDKZIP

Return Value

This function returns GP_NO_ERROR upon success. Otherwise a valid GPResult is returned.

Parameters

connection

[in] A GP connection interface.

productID

[in] The product ID you wish to invite players to.

blocking

[in] GP_BLOCKING or GP_NON_BLOCKING

callback

[in] A user-supplied callback with an arg type of GPFindPlayersResponseArg.

param

[in] Pointer to user-defined data. This value will be passed unmodified to the callback function.

Remarks

This function contacts the Search Manager and attempts to find all profiles that match the search criteria. A profile matches the provided search criteria only if its corresponding values are the same as those provided. Currently, there is no substring matching, and the criteria is case-sensitive.

When the search is complete, the callback will be called.

Section Reference: [Gamespy Presence SDK](#)

See Also: [GPFindPlayersResponseArg](#)