BlackBerry Software Development Kit

Version 2.5

System Utilities API Reference Guide

BlackBerry Software Development Kit 2.5 System Utilities API Reference Guide
Last revised: 18 July 2002

Part number: PDF-04804-001

At the time of publication, this documentation complies with RIM Wireless Handheld version 2.5.

Research In Motion Limited
295 Phillip Street
Waterloo, ON N2L 3W8
Canada

Published in Canada

Contents

About this guide

The System Utilities application programming interface (API) is a new addition to the BlackBerry SDK 2.5. It contains utilities you can use to supplement your applications: the Event Logger, the Registrar, and the Random Number Generator, as well as common string utilities.

For applications based on functions from some components of the BlackBerry SDK (such as the HTTP API and Remote Address Lookup API), implementation of System Utilities API functions is required.

Related documentation

Before you use this guide, you should be familiar with the following documentation. These other resources can help you develop C++ applications for the RIM Wireless Handheld.

- *BlackBerry SDK Developer Guide*

  The *BlackBerry SDK Developer Guide* explains how to use the BlackBerry SDK, with tutorials and sample code to demonstrate how to write handheld applications. For additional information, visit the BlackBerry Developer Zone at `http://www.blackberry.net/developers`.

- `README.txt`

  The `README.txt` file is installed with the BlackBerry Software Development Kit (SDK). It provides information on any known issues and workarounds, as well as last-minute documentation updates and release notes.

Chapter 1
Event Logger API Reference

The Event Logger provides a standardized method for recording events in the handheld's persistent store. The Event Logger API is a new addition to the BlackBerry SDK 2.5.

Events are persistent across a reset of the handheld. The handheld maintains an event queue; when the log gets too full, new events flush old ones out of the queue. You can log events at any of the following six event levels.

Event level	Value	Description
Severe	ELV_SEVERE	log severe events
Error	ELV_ERROR	log error events
Warning	ELV_WARNING	log warning events
Info	ELV_INFO	log info events
Debug	ELV_DEBUG	log debug message events
Always	ELV_ALWAYS	always log events

To be written into the log, each posted event must either have an event level equal to or higher than the current logging level or must be logged with the *Always* event level. For example, if the logging level is set to *Info*, then *Severe*, *Error*, and *Warning* events will be logged, in addition to *Info* events. The default logging level is *Warning*.

To use the Event Logger

1. Include this macro at the beginning of the file containing `PagerMain()`:

   ```
   DECLARE_EVENT_LOGGER
   ```

2. Call `InitEventLogger()` in `PagerMain()`.
3. Include <`iEventLogger.h`> in your application's header file.

iEventLogger

The `iEventLogger` class contains functions for logging events. To log events within an application, include `<iEventLogger.h>` in your code.

Functions

The following functions are listed alphabetically.

The Event Logger API also contains the following inline functions to simplify the logging process.

ClearLog

Clears the Event Logger's database.

```
virtual IRESULT ICALLTYPE ClearLog()
```

Returns Success if the database handle is initialized; error code otherwise.

DBG_LOG

Records an event for debugging on the Event Logger.

```
form 1: inline DBG_LOG(short ID,
    const char * adtlStr = NULL)
form 2: inline DBG_LOG(const char * adtlStr)
```

Parameters

`ID`	The event ID.
`adtlStr`	A NULL-terminated additional string. It should not be very long. The additional string has the following purposes: • used by the viewer if there is a %s in the event string • error string returned by network • make debug logging easier

Description Form 1 saves flash memory by logging IDs; form 2 logs debugging information quickly, without defining event IDs.

GetMinimumLevel

Retrieves the current minimum logging level.

```
virtual IRESULT ICALLTYPE GetMinimumLevel(int * pLevel)
```

Parameters

`pLevel`	A pointer to the current minimum logging level.

Returns The minimum level at which events are logged.

InitEventLogger

Initializes the Event Logger.

```
inline IRESULT InitEventLogger()
```

Returns Success if the Event Logger was successfully initialized.

Description Each application must call this function in `PagerMain()`.

LOG_ERROR

Records the current event on the Event Logger.

```
inline LOG_ERROR(short errID, const char * adtlStr = NULL)
```

Parameters

`errID`	The error level.
`adtlStr`	A NULL-terminated additional string. It should not be very long. The additional string has the following purposes: • used by the viewer if there is a %s in the event string • error string returned by network • make debug logging easier

Description `LOG_ERROR` is an inline function that simplifies the event logging process. It enables you to log an ERROR event without specifying the module name or the event level.

The event level is ERROR by default.

For example, calling `LOG_ERROR` is the equivalent to calling:

```
inline LOG_EVENT(short eventID, EventLever eventLV = ELV_ERROR,  const
char * adtlStr = NULL)
```

LOG_EVENT

Records the current event on the Event Logger.

```
inline LOG_EVENT(short eventID,
    EventLever eventLV,
    const char * adtlStr = NULL)
```

Parameters

`eventID`	The event ID.
`eventLV`	The event level.
`adtlStr`	A NULL-terminated additional string. It should not be very long. The additional string has the following purposes: • used by the viewer if there is a %s in the event string • error string returned by network • make debug logging easier

Description `LOG_EVENT` is an inline function that simplifies the error logging process. It enables you to log an event without specifying the module name.

LOG_INFO

Records the current event on the Event Logger.

```
inline LOG_INFO(short evtID,
    const char * adtlStr = NULL)
```

Parameters

`evtID`	The event ID.
`adtlStr`	A NULL-terminated additional string. It should not be very long. The additional string has the following purposes: • used by the viewer if there is a %s in the event string • error string returned by network • make debug logging easier

Description `LOG_INFO` is an inline function that simplifies the event logging process. It enables you to log an INFO event without specifying the module name or the event level.

The event level is INFO by default.

For example, calling `LOG_INFO` is the equivalent to calling:

```
inline LOG_EVENT(short eventID, EventLever eventLV = ELV_INFO,  const char
* adtlStr = NULL)
```

LogEvent

Records the current event on the Event Logger.

```
Form 4: virtual IRESULT ICALLTYPE LogEvent(
    const char * pModuleName,
    unsigned short EventID,
    EventLever EventLV,
    const char * pAdditionalStr = NULL )
```

Parameters

`pModuleName`	The `VersionPtr` for the application. It is defined in `PagerMain()` and registers the application with the OS task switcher. For more information on `VersionPtr`, refer to the *BlackBerry SDK Developer Guide*.
`EventID`	The event code. Each application has its own set of code.
`EventLV`	The error level.
`pAdditionalStr`	A NULL-terminated additional string. It should not be very long. The additional string has the following purposes: • used by the viewer if there is a %s in the event string • error string returned by network • make debug logging easier

Returns Success if the event was successfully logged.

Description If you do not specify an `EventLV` parameter, the event is always logged.

SetMinimumLevel

Sets the current minimum logging level.

```
virtual IRESULT ICALLTYPE SetMinimumLevel(int level)
```

Parameters	`level`	The new minimum logging level.

Returns Success if the new logging level has been saved in the database.

Description This is the minimum level that must be returned to log the event in the database.

Error codes

Event Logger functions return an IRESULT code.

IRESULT	Code	Description
IR_EL_SUCCESS	0	The operation completed successfully.
IR_EL_FAILED	-1	The operation failed.
IR_EL_NO_LOGGER	-2	An instance of the Event Logger could not be created.
IR_EL_NOT_READY	-3	The Event Logger has not been instantiated.
IR_EL_NOT_LOG_LEVEL	-4	The application specified an non-existant logging level.

iEventViewer

`iEventViewer` provides an interface that the other DLLs can use to access the Event Viewer from their own threads.

To display event logs within an application, include `<iEventViewer.h>` in your code.

Functions

DisplayEvents

Displays event logs on the handheld.

```
virtual IRESULT ICALLTYPE DisplayEvents(const char * pModuleName = NULL)
```

Parameters

`pModuleName`	The module for which you are logging events, identified by the `VersionPtr` of the application. It is defined in `PagerMain()` and registers the application with the OS task switcher. For more information on `VersionPtr`, refer to the *BlackBerry SDK Developer Guide*.

Description If you do not specify a module name, the Event Viewer displays all system events.

Chapter 2 Registrar API Reference

The Registrar is an application that manages registration and instantiation of objects which implement interface-based APIs.

The Registrar enables you to instantiate an object with an interface pointer. It is applicable to many applications, but particularly to several of the BlackBerry SDK APIs. In the scope of the HTTP API, the Registrar enables protocols to be opened and registered, and manages wireless connections between a handheld and the Internet. In the scope of the Remote Address Lookup API, the Registrar instantiates query objects and manages Address Book referencing to the results.

The Registrar does not provide an explicit function to terminate an instance of an object. Instead, the `iBase` interface enables you to manage references to objects; when the reference count on an object reaches zero, the object is terminated automatically.

The `iStr` and `iPtr` classes manage string memory and object lifetimes, respectively.

About interface-based APIs

Interface-based APIs (in contrast to exported classes) provide the following benefits:

- **Implementation improvement without breaking binary compatibility**.

  Implementing constructors in the object (server) code enables the implementation (new members and virtual methods) to be modified without having to recompile the client code.
- **Client and server code have no static dependencies**.

  A lack of static dependencies in the code prevents cyclic dependencies between applications.
- **Client code can test for the existence of an API**.

  If a desired API does not exist, an application can continue the method normally without using the missing API.
- **Enables alternative implementations**.

  The implementation of an API can be replaced with an alternative implementation by loading different applications onto the handheld.

Interfaces/Classes	Page	Header file
iBase	14	iBase.h
iStr	16	iStr.h

iBase

`iBase` is used to instantiate an object with an interface pointer. This enables you to create an instance of an object without a static dependancy on an interface; the object can be referenced through the Registrar, rather than the client that originally created it. The object is assigned a unique ID.

When an object is instantiated, the reference count is incremented to it. The reference count on an object is used to determine how many clients are currently referencing it. When the reference count on an object (such as a stream or connection) decrements to zero, it is closed automatically.

`iBase` is the base interface for all interface-based APIs. To include System Utilities API functions in your application, you must include `<iBase.h>` in your code.

Functions

The following functions are listed alphabetically.

iBase::AddRef

Increments the reference count to the object by one.

```
virtual uint ICALLTYPE AddRef() = 0
```

Returns The current (incremented) reference count.

iBase::QueryInterface

Casts and sets the `iface` pointer to an object as specified by the Interface ID.

```
virtual IMETHOD QueryInterface(InterfaceId iid, void ** iface) = 0
```

Parameters

`iid`	The interface with which the `iface` pointer is cast.
`iface`	The pointer to the object to instantiate.

Returns

```
IRESULT_SUCCESS
IRESULT_NULL_POINTER
IRESULT_NO_INTERFACE
```

Description `QueryInterface` instantiates an object with an interface pointer.

Additionally, `QueryInterface` increments the reference count to the specified object (that is, it raises the reference count from zero to one).

iBase::Release

Decrements the reference count to the object by one. When the reference count to a connection reaches zero, it is closed.

```
virtual uint ICALLTYPE Release() = 0
```

Returns The current (decremented) reference count.

iStr

`iStr` manages memory allocated for string objects. It allows for allocated memory to be freed by the calling application. By assigning a parameter using `iStr`, you can manage the memory allocated to the object.

The iStr constructor has four forms:

```
Form 1: iStr()
Form 2: iStr(int size)
Form 3: iStr(const char * sz)
Form 4: iStr(const iStr & that)
```

Parameters

`size`	The length of the buffer.
`sz`	A pointer to the string for this object to contain.
`that`	A reference to an already initialized `iStr` object.

Description Form 2 creates an `iStr` object with a specified buffer size; Form 3 creates an `iStr` object with a pointer to the string to be set; Form 4 creates an `iStr` object that is a duplicate of `that`.

Functions

The following functions are listed alphabetically.

iStr::~iStr

Destroys an instance of an `iStr` object.

```
~iStr()
```

iStr::Append

Appends a string to the end of the existing buffer.

```
Form 1: bool Append(const char * sz)
Form 2: bool Append(const char * sz, uint length)
```

Parameters

`sz`	A pointer to the string to append to the buffer.
`length`	The length of the string.

Description The length of the string to be appended can be specified. The internal allocated memory is re-allocated in 16k blocks as required to fit the resulting string.

Form 1 omits the string length argument. Form 2 specifies the length of the string to append to the buffer.

iStr::AppendExact

Appends a string to the end of the existing buffer.

```
Form 1: bool AppendExact(const char * sz)
Form 2: bool AppendExact(const char * sz, uint length)
```

Parameters

`sz`	A pointer to the string to append to the buffer.
`length`	The length of the string.

Description The length of the string to be appended can be specified. The internal allocated memory is re-allocated to the exact size of the resulting string.

Form 1 omits the string length argument. Form 2 specifies the length of the string to append to the buffer.

iStr::Empty

Empties the internal buffer.

```
void Empty()
```

iStr::IsEmpty

Determines if the internal buffer is empty.

```
bool IsEmpty()
```

Returns True if the buffer is empty; false otherwise.

iStr::Grow

Increases the internal RAM buffer without appending or setting a string.

```
bool Grow(uint newSize)
```

Parameters

`newSize`	The size to extend the buffer length to.

iStr::operator

```
Form 1: operator char*() const
Form 2: iStr& operator=(const char * sz)
Form 3: iStr& operator=(const iStr & that)
Form 4: iStr& operator+=(char * sz)
```

Parameters

`sz`	The length of the buffer.
`that`	A reference to an already initialized `iStr` object.

Description Form 2 sets the left side parameter to be a duplicate of `sz`. Form 3 sets the left-side parameter to be a duplicate of `that`. Form 4 appends `sz` to the left-side parameter.

iStr::Set

Clears the existing buffer and sets a string to the empty buffer.

```
Form 1: bool Set(const char * sz)
Form 2: bool Set(const char * szStart, uint length)
```

Parameters

`sz`	A pointer to the string to append to the buffer.
`szStart`	A pointer to the string to append to the buffer.
`length`	The length of the string.

Description The length of the string can be specified. The internal allocated memory is re-allocated in 16k blocks as required to fit the resulting string. For example, a 20k string would be allocated 32k.

Form 1 omits the string length argument. Form 2 specifies the length of the string to set in the buffer.

iStr::SetExact

Clears the existing buffer and sets a string to the empty buffer.

```
Form 1: bool SetExact(const char * sz)
Form 2: bool SetExact(const char * sz, uint length)
```

Parameters

`sz`	A pointer to the string to append to the buffer.
`length`	The length of the string.

Description The length of the string can be specified. The internal allocated memory is re-allocated to the exact size of the resulting string. For example, a 20k string would be allocated 20k.

Form 1 omits the string length argument. Form 2 specifies the length of the string to set in the buffer.

Chapter 3
RNG API Reference

The RNG API defines random number generation rountines for RIM Wireless Handhelds. To generate random number data within an application, include `<RNG.h>` in your code.

Functions

rand

Fills a buffer with random bytes.

```
MessageDllAccess void rand(
    void * buffer,
    int length)
```

Parameters

buffer	A buffer to contain the random bytes
length	The length of `buffer`.

Description Bits are random in each byte.

seed

Seeds the random number generator.

```
MessageDllAccess void seed(
    void const * seedData,
    int length)
```

Parameters	seedData	A series of random bytes.
	length	The number of random bytes in `seedData`.

Description `Seed` initializes the random number generator.

Chapter 4 String Utilities API Reference

The String Utilities API provides common utilities that are not available through the standard C library, including string-handling routines. These utility routines are used by both the UI engine and by applications.

See the *BlackBerry SDK Developer Guide* for a list of standard C functions that can and cannot be used when writing applications for the RIM Wireless Handheld.

The functions in the String Utilities API are defined in `utilities.h`; the library is `utilities.lib`.

Functions

The following functions are listed alphabetically.

atoi

Converts a string to an integer.

```
inline int atoi(
    const char * buffer,
    int radix = 10)
```

Parameters

`buffer`	A pointer to the string to convert to an integer.
`radix`	One of: • 8 - set to convert the string to octal. • 10 - set to convert the string to decimal. • 16 - set to convert the string to hexademical.

Returns The string as an integer.

Description `atoi` converts a string to an integer. The string must represent an integer (that is, consist of a series of numeric digits, with an optional operator sign). `atoi` continues converting until a non-numeric digit is reached, at which point it returns the converted integer.

pattern_match

Determines if a string matches a simple pattern, ignoring case.

```
bool pattern_match(const char * text,
    const char * pattern)
```

Parameters

`text`	The text to compare to `pattern`.
`pattern`	The pattern that `text` must match.

Returns True if `text` matches `pattern`; false otherwise.

Description `pattern_match` is a simple routine to determine if one string matches another. It is case-insensitive and ignores spaces.

prefix_match

Determines if a string begins with a specific prefix, considering case.

```
bool prefix_match(
    char const * string,
    char const * prefix)
```

Parameters

`string`	The text to test if it begins with `prefix`.
`prefix`	The text that `string` must begin with.

Returns True if `string` begins with `prefix`; false otherwise.

Description `prefix_match` considers case when comparing the strings.

For example, if `string` is *Smith* and `prefix` is *Sm*, `prefix_match_i` returns true.

prefix_match_i

Determines if a string begins with a specific prefix, ignoring case.

```
bool prefix_match_i(
    char const * string,
    char const * prefix)
```

Parameters

`string`	The text to test if it begins with `prefix`.
`prefix`	The text that `string` must begin with.

Returns True if `string` begins with `prefix`; false otherwise.

Description `prefix_match_i` compares two strings after converting them to lower case.

For example, if `string` is *Smith* and `prefix` is *sm*, `prefix_match_i` returns true.

RimSmartStrcmp

Compares two strings.

```
int RimSmartStrcmp(
    const char * str1,
    const char * str2)
```

Parameters

`str1`	The first string to compare.
`str2`	The second string to compare.

Returns An integer that is:

- < 0 if `str1` precedes `str2` alphabetically.
- 0 if `str1` and `str2` are considered equal.
- > 0 if `str1` follows `str2` alphabetically.

Description `RimSmartStrcmp` compares two strings after converting them to lower case, and removing any accents. If this does not resolve a difference, the original case of the strings is considered. If the strings are still equals, the original accents of the strings (if any) are compared.

For example, if `string1` is equal to *Smith* and `string2` is equal to *smith*, `RimSmartStrcmp` returns a negative integer.

RimStricmp

Compares two strings, ignoring case.

```
int RimStricmp(
    const char * str1,
    const char * str2)
```

Parameters

`str1`	The first string to compare.
`str2`	The second string to compare.

Returns An integer that is:

- < 0 if `str1` precedes `str2` alphabetically.
- 0 if `str1` and `str2` are considered equal.
- > 0 if `str1` follows `str2` alphabetically.

Description `RimStricmp` compares two strings after converting them to lower case.

For example, if `string1` is equal to *Smith* and `string2` is equal to *smyth*, `RimStricmp` returns a negative integer.

RimStristr

Searches for the first instance of a substring pattern within a string, ignoring case.

```
char * RimStristr(const char * text,
    const char * pattern,
    int text_length,
    unsigned char * skip,
    int pattern_length = -1)
```

Parameters

`text`	The text to search for `pattern`.
`pattern`	The substring pattern to search `text` for.

Parameters

`text`	The text to search for `pattern`.
`text_length`	The length of the string in `text`.
`skip`	A portion of `text` not to search.
`pattern_length`	The length of the string in `pattern`.

Returns A pointer to the instance of the substring pattern within the string.

Description `RimStristr` searches for the first instance of a substring within a string. If the pattern cannot be located, it converts `text` and `pattern` to lower case and searches for the pattern again.

RimStristr_init

Determines the length of a pattern, ignoring case.

```
int RimStristr_init(unsigned char * skip,
    const char * pattern,
    int pattern_length = -1)
```

Parameters

`skip`	A portion of the pattern to ignore.
`pattern`	The pattern to determine the length of.
`pattern_length`	Should be set to -1.

Returns The length of the pattern.

RimStristrTerm

Determines the length of a terminating substring within a string, ignoring case.

```
#define RimStristrTerm(a,b,c)
    RimStristr((a),(b),((c)-(a))
    )
```

Parameters

`a`	The substring to search for.
`b`	The length of the substring.
`c`	A portion of the text to skip.

Returns The length of the terminating substring.

RimStrstr

Searches for the first instance of a substring pattern within a string.

```
char * RimStrstr(const char * text,
    const char * pattern,
    int text_length,
    unsigned char * skip,
    int pattern_length = -1)
```

Parameters

`text`	The text to search for `pattern`.
`pattern`	The pattern to search `text` for.
`text_length`	The length of the string in `text`.
`skip`	A portion of `text` not to search.
`pattern_length`	The length of the string in `pattern`.

RimStrstr_init

Determines the length of a pattern.

```
int RimStrstr_init(unsigned char * skip,
    const char * pattern,
    int pattern_length = -1)
```

Parameters

`skip`	A portion of the pattern to ignore.
`pattern`	The pattern to determine the length of.
`pattern_length`	Should be set to -1.

Returns The length of the pattern.

RimStrtol

Converts a string to a signed long integer.

```
long RimStrtol(const char * nptr,
    const char ** endptr,
    int ibase)
```

Parameters

`nptr`	A pointer to the string to convert.
`endptr`	A pointer to the position in the string where conversion ended (that is, the next character after the last numeric digit in the string.) This is a result parameter.
`ibase`	The conversion base. One of: • `0x` or `0X` - if specified, digits are treated as hexadecimal. • `0` - if specified, digits are treated as octal. • `1` to `9` - if specified, digits are treated as decimal.

Returns The string as a signed longer integer.

RimStruicmp

Compares two strings, ignoring case and accents.

```
int RimStruicmp(
    const char * str1,
    const char * str2)
```

Parameters

`str1`	The first string to compare.
`str2`	The second string to compare.

Returns An integer that is:

- < 0 if `str1` precedes `str2` alphabetically.
- 0 if `str1` and `str2` are considered equal.
- > 0 if `str1` follows `str2` alphabetically.

Description `RimStruicmp` compares two strings after converting them to lower case, and removing any accents.

RimStrucmp

Compares two strings, ignoring any accents.

```
int RimStrucmp(
    const char * str1,
    const char * str2)
```

Parameters

`str1`	The first string to compare.
`str2`	The second string to compare.

Returns An integer that is:

- < 0 if str1 precedes str2 alphabetically.
- 0 if str1 and str2 are considered equal.
- > 0 if str1 follows str2 alphabetically.

Description RimStrucmp compares two strings after removing any accents.

strcat

Concatenates two strings.

```
char * strcat(
    char * dest,
    int dest_length,
    const char * src)
```

Parameters

dest	The string to append src to.
dest_length	The maximum length to permit dest to be.
src	The string which will be appended to dest.

Returns A pointer to the concatenated string.

Description strcat appends the contents of src to dest.

strcpy

Copies a string.

```
char * strcpy(char * dest,
    int dest_length,
    const char * src)
```

Parameters

dest	A pointer to the copy destination.
dest_length	The length of the destination string.
src	A pointer to the source string.

Description strcpy copies the contents of src into dest. dest_length should be long enough to hold the contents of src.

strncpy

Copies part of a string.

```
char * strncpy(char * dest,
    int dest_length,
    const char * src,
    int src_length)
```

Parameters

`dest`	A pointer to the copy destination.
`dest_length`	The length of the destination string.
`src`	A pointer to the portion of the source string.
`src_length`	The length of the portion of the source string.

Description `strncpy` copies the first `src_length` number of characters from `src` to `dest`. `dest_length` should be long enough to hold on the contents of `src`.

`strncpy` is independant of the UI Engine.

strncmp

Compares part of two strings.

```
int strncmp(
    const char * str1,
    const char * str2,
    int len)
```

Parameters

`str1`	The first string portion to compare.
`str2`	The second string portion to compare.
`len`	The length of the string portions to compare.

Returns An integer that is:

- < 0 if `str1` is shorter than `str2`.
- 0 if `str1` and `str2` are equal in length.
- > 0 if `str1` is longer than `str2`.

Description `strncmp` compares a specific portion (`len`) of two strings.

strnicmp

Compares part of two strings, ignoring case.

```
int strnicmp(
    const char * str1,
    const char * str2,
    int len)
```

Parameters

`str1`	The first string to compare.
`str2`	The second string to compare.
`len`	The length of the string portions to compare.

Returns An integer that is:

- < 0 if `str1` precedes `str2` alphabetically.
- 0 if `str1` and `str2` are considered equal.
- > 0 if `str1` follows `str2` alphabetically.

Description `strnicmp` compares a specific portion (`len`) of two strings after converting them to lower case. This function depends on the UI Engine.

Index of functions

Index