

BlackBerry Software Development Kit

Version 2.5

Operating System API Reference Guide

BlackBerry Software Development Kit Version 2.5
Operating System API Reference Guide
Last modified: 24 May 2002

Part number: PDF-04637-001

At the time of publication, this documentation complies with BlackBerry Software Development Kit version 2.5.

© 2002 Research In Motion Limited. All Rights Reserved. The BlackBerry and RIM families of related marks, images and symbols are the exclusive properties of Research In Motion Limited. RIM, Research In Motion, 'Always On, Always Connected', the "envelope in motion" symbol and the BlackBerry logo are registered with the U.S. Patent and Trademark Office and may be pending or registered in other countries. All other brands, product names, company names, trademarks and service marks are the properties of their respective owners.

The handheld and/or associated software are protected by copyright, international treaties and various patents, including one or more of the following U.S. patents: 6,278,442; 6,271,605; 6,219,694; 6,075,470; 6,073,318; D445,428; D433,460; D416,256. Other patents are registered or pending in various countries around the world. Visit www.rim.net/patents.shtml for a current listing of applicable patents.

While every effort has been made to ensure technical accuracy, information in this document is subject to change without notice and does not represent a commitment on the part of Research In Motion Limited, or any of its subsidiaries, affiliates, agents, licensors, or resellers. There are no warranties, express or implied, with respect to the content of this document.

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

Published in Canada

Contents

About this guide	5
Related documents	5
System API	7
Structures	7
Thread and communication functions.....	12
Memory allocation functions	24
Time functions.....	26
User notification functions	30
Password functions	33
Miscellaneous functions.....	34
File System API	47
Structures	47
Functions.....	48
Return codes	67
Serial communications API	69
Serial communications functions	69
Constants.....	75
Error Codes.....	75
Events	76
Keypad API	77
Functions.....	77
Constants.....	79
Events	79

LCD API.....	81
Structures	81
LCD functions	82
Constants.....	104
Return codes	105
Peripheral API.....	107
Functions.....	107
Device events	111
SYSTEM device events.....	112
TIMER device events	114
RTC (real-time clock) device events.....	114
HOLSTER device events.....	115
KEYPAD device events.....	115
RADIO device events.....	118
COM device events	119
Index of functions	121
Index	123

About this guide

This guide provides detailed reference information for the Operations System application programming interface (API). The OS API can manage multiple applications running simultaneously. Applications can call each other, enabling you to write new components for the handheld without needing to rewrite entire applications.

Related documents

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>.

About this guide

Chapter 2

System API

This section provides information on the System API, declared in `Rim.h`:

- structures (refer to page 7)
- functions (refer to page 12)

Structures

The following set of structures are specific to the System API.

AlertConfiguration	7
DEVICE_INFO	9
MESSAGE	10
PID	11
TIME	11

AlertConfiguration

The `AlertConfiguration` structure stores information about the handheld notification settings.

```
typedef struct {  
    BYTE    duty;  
    BYTE    volume;  
    BYTE    maxVolume;  
    BYTE    repetitions;  
    BYTE    vibrateTime;  
    BYTE    vibrateType;  
    BYTE    inHolsterNotify;  
    BYTE    outOfHolsterNotify;  
    BYTE    pause;  
} AlertConfiguration;
```

Chapter 2: System API

Field	Values	Description
duty	1 to 99	Percentage ratio of the on/off frequency pulse
volume	1 to 100	Initial beep volume, from 1 (softest) to 100 percent (loudest)
maxVolume	1 to 100	Maximum beep volume, from 1 to 100 percent. If the number of repetitions is greater than one, the volume increases on each repetition, with the last repetition at maxVolume.
repetitions	1 to 10	The number of times TONE_SEQUENCE_X is sounded, where X is the tone number.
vibrateTime	0 to 255	The vibrate time, in tenths of a second (i.e. from 0 seconds to 25.5 seconds)
vibrateType	PULSE_VIBRATE	The vibration repeats the following cycle for a total of vibrateTime/10 seconds: ON for ¼ second OFF for ¼ second ON for ¼ second OFF for 1 second
	CONTINUOUS_VIBRATE	Vibration is turned on for vibrateTime/10 seconds
inHolsterNotify outOfHolsterNotify	These two variables allow different tone/vibrator strategies to be selected, depending on whether the handheld is in its holster or not. The following strategies are available:	
	NO_NOTIFY	Do not vibrate or sound any tones.
	TONE_ALERT	Sound tones only; do not use vibrator.
	VIBRATE_ALERT	Vibrate only; do not sound any tones.

Field	Values	Description
	VIBRATE_AND_TONE	Use both vibrator and tones simultaneously to alert the user.
	VIBRATE_THEN_TONE	At first, vibrate only. If the alert is not cancelled after the vibration time, sound the tone sequence. Each repetition of the tone sequence is sounded at a louder volume, as described above, until the alert is cancelled or the maximum number of repetitions is reached.
pause	0 to 255	Tenths of a second of pause time between repetitions of a tone sequence.

DEVICE_INFO

The DEVICE_INFO structure stores information about the user's handheld.

```
typedef struct {
    BYTE    deviceType;
    BYTE    networkType;
    WORD    reserved1;
    union {
        DWORD    MAN;
        DWORD    LLI;
        BYTE    reserved2[16];
    } deviceId;
    DWORD    ESN;
    BYTE    HSN[16];
    BYTE    reserved3[24];
} DEVICE_INFO;
```

Field	Description
deviceType	Valid values are: DEV_PAGER DEV_OEM DEV_HANDHELD
networkType	Valid values are: NET_MOBITEX NET_DATATAC
reserved1	Reserved for future expansion.
DeviceId	The identifier (traditionally MAN or LLI number) of the handheld.

Chapter 2: System API

Field	Description
ESN	The electronic serial number.
HSN[16]	Reserved for future expansion.
reserved3	Reserved for future expansion.

MESSAGE

The MESSAGE structure represents a system or application message. Handheld applications receive external event notification from the system through MESSAGEs. Applications can also use MESSAGEs to communicate between their own threads or with other applications.

```
typedef struct {  
    DWORD Device, Event, SubMsg, Length;  
    char * DataPtr;  
    DWORD Data[2];  
} MESSAGE;
```

The Device and the Event fields are mandatory. A number of calls require the application to examine the contents of Data[0]. However, the OS does not examine messages between applications, so use of the fields in the MESSAGE structure is up to each individual developer.

Field	Description
Device	The sender of the message (mandatory). If the sender is a Wireless Handheld, the device field of the message is set to one of: DEVICE_SYSTEM DEVICE_KEYPAD DEVICE_RTC DEVICE_TIMER DEVICE_COM1 DEVICE_RADIO DEVICE_HOLSTER DEVICE_USER Values of DEVICE_USER or greater can be used by user applications; see <code>rim.h</code> for values. See "Device events" on page 111 for a description of each device and its events.
Event	The event that caused this message to be sent (mandatory)
SubMsg	Extra data pertaining to the event (optional)

Field	Description
Data	Extra data pertaining to the event (optional)
DataPtr	A reference external to the message, used to pass large amounts of data (optional)
Length	The record size referenced by DataPtr, if used (optional)

PID

The PID structure stores information about a process.

```
typedef struct {
    const char * Name;
    BOOL EnableForeground;
    const BitMap * Icon;
} PID;
```

Field	Description
Name	The name associated with the process.
EnableForeground	Used by the task-switcher to determine if the process in question is foregroundable or not. Processes that provide no user interface (like background tasks) should not appear as a foreground task and should have this field set to FALSE.
Icon	A pointer to the tasks icon. The icon is displayed by the task switcher, if the EnableForeground field is TRUE.

TIME

```
typedef struct    {
    BYTE    second;
    BYTE    minute;
    BYTE    hour;
    BYTE    date;
    BYTE    month;
    BYTE    day;
    WORD    year;
    BYTE    TimeZone;
} TIME;
```

Member	Meaning
second	0 to 59
minute	0 to 59
hour	0 to 23; the internal clock uses 24 hour time format
date	0 to 31, depending on the month
month	An enumerated field. January = 1, February = 2, ..., December = 12
day	An enumerated field. Sunday = 0, Monday = 1, ..., Saturday = 6
year	The valid range of years is between 1998 and 2090
TimeZone	An enumerated field. GMT is 0; all other zones are counted in half hours from that point. For example, EST is GMT -5 hours which would be stored as -10 half hours

Thread and communication functions

This section describes thread and inter-process communication functions, listed in alphabetical order.

RimCreateThread	13
RimDisableAppSwitch	13
RimEnableAppSwitch	13
RimFindTask	14
RimGetCurrentTaskID	14
RimGetForegroundApp	14
RimGetMessage	14
RimGetPID	16
RimInvokeTaskSwitcher	16
RimPeekMessage	16
RimPostMessage	17
RimRegisterMessageCallback	17
RimReplyMessage	19
RimRequestForeground	20
RimSendMessage	20
RimSendSyncMessage	20
RimSetPID	21
RimSetReceiveFromDevice	21
RimTaskYield	22
RimTerminateThread	22
RimToggleMessageReceiving	23
RimWaitForSpecificMessage	23

RimCreateThread

Enables an application to dynamically create a new thread, with its own stack, in the application server.

```
TASK RimCreateThread(
    void (*Entry)(void),
    DWORD Stacksize)
```

- Parameters**
- | | |
|-----------|---|
| Entry | A pointer to the entry function. |
| Stacksize | The size, in bytes, of the local stack for the newly created thread. This value must be large enough to hold the stack for the thread, plus the stack space required by the API when called. Setting this value too small creates unpredictable results. It is recommended that this value never be set less than 2000. |
- Returns** The task handle of the newly created thread; 0 (zero) if the thread could not be created.
- Description** The new thread shares the same data space as the parent process. If there are no more task handles available or not enough memory then the function fails. The thread can be initialized by the attributes below.
- Threads that are created cannot be placed on the foreground unless the thread enables this attribute with the `RimSetPID` function.
- Created threads do not receive any radio events unless they register for Radio events using the `RadioRegister()` function.
- For example, see `bouncer.c` and `ping.c`.

RimDisableAppSwitch

Prevents other applications from gaining foreground processing.

```
BOOL RimDisableAppSwitch()
```

- Returns** True if the function was successful; false otherwise.
- Description** This function, when called by the foreground program, prevents other applications from gaining foreground processing. This function should be used when entering a critical procedure in an application that must remain in the foreground. Only the task which called `RimDisableAppSwitch` can re-enable task-switching by calling `RimEnableAppSwitch`.

RimEnableAppSwitch

Enables other applications to request foreground processing.

```
BOOL RimEnableAppSwitch()
```

- Returns** True if the function was successful; false otherwise.

Chapter 2: System API

Description This function enables other applications to request foreground processing. Only the task which called `RimDisableAppSwitch` can re-enable task-switching by calling `RimEnableAppSwitch`.

RimFindTask

Enables an application to search for a task based on its version string.

```
TASK RimFindTask(char * Prefix)
```

Parameters `Prefix` The prefix of the version string of the task to search for.

Returns The handle of the task that has a version string that starts with `prefix`. If no such task is found, `RimFindTask` returns `TASK_NOT_FOUND`.

Description This function enables an application to search for a task based on its version string. The first task found that has a version string that starts with the string specified by `prefix` is selected. Threads created with `RimCreateThread` can also be found in this manner, provided they have used `RimSetPID` to set their version string.

RimGetCurrentTaskID

Retrieves the current task ID.

```
TASK RimGetCurrentTaskID()
```

Returns The task handle of the currently running task.

Description The handle of the current running task. This task handle is the same value when a task receives the `INITIALIZE` and `POWER_UP` events from the application server.

For example, see `bouncer.c`.

RimGetForegroundApp

Retrieves the handle of the application that currently has focus.

```
TASK RimGetForegroundApp()
```

Returns The task handle of the process currently running in the foreground.

Description This function is used when multiple applications are running on the handheld. Applications running in the background can request the handle of the application that currently has focus.

RimGetMessage

Enables the calling application to obtain the next message from its message queue.

```
TASK RimGetMessage(MESSAGE * Msg)
```

Thread and communication functions

Parameters `Msg` A pointer to a MESSAGE structure that receives the message.

Returns The task handle of the task that sent or posted the received message.

Description This function enables the calling application to obtain the next message from its message queue. If no messages are available to the calling task, the task is suspended until a message becomes available.

It is important that a task performs `RimGetMessage` to obtain events, as this enables the application server to determine which tasks are currently not using the CPU. The processor is then put into low power mode.

For example, see `bouncer.c`, `hello.c`, `clock.c`, `comtest.c`, `ping.c`, `realtime.c`, and `regmess.c`.

RimGetPID

Enables a process to inquire about the attributes of other processes.

```
BOOL RimGetPID(  
    TASK hTask,  
    PID * Pid,  
    const char ** Subtitle)
```

Parameters	hTask	The task number for which display information is desired. Valid values are 0 (the system task) through (MAX_APPLICATIONS -1).
	Pid	A pointer to the pid structure to be filled in with information about the specified task.
	Subtitle	The address of the pointer to be set to point to the specified task's subtitle string.

Returns True if the task exists or false if the task number is invalid.

Description This function enables a process to inquire about the attributes of other processes. This enables an application to create its own task-switching menu . If either the `Pid` or `Subtitle` parameter is NULL, it is not used.

RimInvokeTaskSwitcher

Brings the Switcher menu to the foreground.

```
void RimInvokeTaskSwitcher()
```

Description This function brings the Switcher menu to the foreground, which allows the user to switch to another task.

RimPeekMessage

Enables a task to determine whether there are any messages available for it to process.

```
BOOL RimPeekMessage()
```

Returns True if there is a message waiting on the event queue for the calling task; false otherwise.

Description This function enables a task to determine whether there are any messages available for it to process. A foreground task should call `RimPeekMessage` to handle keypad or radio messages while it is performing a long operation.

RimPostMessage

Enables applications to post messages with the application server to be delivered to other applications.

```
void RimPostMessage(
    TASK hTask,
    const MESSAGE * Msg)
```

Parameters

hTask	An application handle to which the message is being sent.
Msg	A pointer to a message structure.

Description This function enables applications to post messages with the application server to be delivered to other applications. This function returns immediately without waiting for the message to be processed by the other application. Attempting to send messages to tasks that do not exist generates an exception. Also see RimSendMessage for synchronously sending a message to another task.

For example, see `bouncer.c`.

RimRegisterMessageCallback

Registers a callback function to be called upon retrieval of certain message types.

```
BOOL RimRegisterMessageCallback(
    DWORD MessageBits,
    DWORD MaskBits,
    CALLBACK_FUNC HandlerFunc)
```

Chapter 2: System API

Parameters	MessageBits	This parameter contains the event number for which you want to register. All system events also contain a copy of the device ID in bits 8 to 15.
	MaskBits	This parameter contains a bitmask of which bits of <code>MessageBits</code> must mask the event number in order for the callback to take place. Setting this value to <code>0xffffffff</code> causes the callback to be made only on the exact event. Setting it to zero causes the callback to be made on every event, while setting it to <code>0xff00</code> causes the callback to happen on every event from a specific device.
	HandlerFunc	A pointer to a function to process the message. If this parameter is set to <code>NULL</code> , the function registered for that message is unregistered. The function must be declared as follows: <pre>int Handler(MESSAGE * msg)</pre> This function is called when the callback criteria is met. The function is always called when the application is in a state of blocking on <code>RimGetMessage</code> . The call is made in the registered application's context.

If the handler function returns true, the message is passed on to subsequent handler functions. If the handler function returns false, the message is considered processed and is not forwarded. If the handler function modifies the message, then the modified message is subsequently forwarded to other handlers and `RimGetMessage`.

Multiple handler functions can process the same message. Re-registering a new function for the same event does not cancel the previous registration. If multiple handler functions examine the same message, the most recently registered handler function is called first. To cancel a previous registration, register `NULL` as the function pointer first.

Returns True if successful, otherwise false. The function fails if there is no available memory.

Description Registering a callback message enables for certain functions to be called on certain events without having to check for these types of events on every call to `RimGetMessage`.



Note: Registering callbacks is application-specific. Registering callbacks for certain events does not affect the processing of messages in other applications. However, calls to `RimGetMessageRaw` can override the callback. Refer to `RimGetMessageRaw` (page 39).

For example, see `regmess.c`.

RimReplyMessage

Responds to applications that are waiting for a return message by calling `RimSendSyncMessage`.

```
BOOL RimReplyMessage(  
    TASK HTask,  
    MESSAGE * Msg)
```

Parameters

HTask	The task handle of the application to which the message is to be sent.
Msg	The pointer to the message to be returned to the task that called <code>RimSendSyncMessage</code> .

Returns True if the message is sent to a valid task and the task is waiting for a `RimReplyMessage` from the calling task; false otherwise.

Description The `RimReplyMessage` function is used to respond to applications that are waiting for a return message by calling `RimSendSyncMessage`. It is the responsibility of the receiving application to call `RimReplyMessage` or the sending application will never unblock.

RimRequestForeground

Enables background applications to be switched into the foreground.

```
BOOL RimRequestForeground(TASK HTask)
```

Parameters HTask The task handle requesting foreground.

Returns True if the request was granted or false if the request failed.

Description This function enables background applications to be switched into the foreground. If a request is granted, the current foreground application is notified by a SWITCH_BACKGROUND event. If a task requests foreground on behalf of another task, that task receives the SWITCH_FOREGROUND event.



Note: This function can be disabled and enabled. See the RimDisableAppSwitch and RimEnableAppSwitch functions for more information.

For example, see `bouncer.c`.

RimSendMessage

Enables applications to send messages synchronously to other applications.

```
BOOL RimSendMessage(  
    TASK HTask,  
    MESSAGE * Msg)
```

Parameters HTask The task handle of the application which receives the message.

Msg A pointer to a MESSAGE structure.

Returns True if the message could be sent to the destination task; 0 (zero) if the destination task was in a state where it could not receive a message immediately. True can only be returned after the destination task has received the message. It is not guaranteed that the receiving task is the first or only task that will run before RimSendMessage returns.

Attempting to send a message to a task that is not available generates an exception.

Description This function enables applications to send messages synchronously to other applications. This function does not return until the destination task has received the message. Tasks cannot synchronously send messages to themselves. Use the asynchronous RimPostMessage instead. Refer to the *Developer Guide* for more information on asynchronous and synchronous sends.

RimSendSyncMessage

Enables an application to send messages to other applications synchronously and wait for a reply.

```

BOOL RimSendSyncMessage(
    TASK HTask,
    MESSAGE * Msg,
    MESSAGE * ReplyMsg)

```

Parameters

HTask	The task handle of the application to which the message is to be sent.
Msg	A pointer to the message to be sent.
ReplyMsg	A pointer to a MESSAGE structure that will be populated by the reply message from the receiving task.

Returns True if the message is sent to a valid task, that task receives the message and does a `RimReplyMessage` to the calling process. If the task being sent to is invalid, or is terminated during the `RimSendSyncMessage` then false is returned. When false is returned, the value placed in `ReplyMsg` is undefined.

Attempting to send a message to a task that is unavailable generates an exception.

Description The `RimSendSyncMessage` function enables an application to send messages to other applications synchronously and wait for a reply. This function does not return successfully until the message has been received and the receiver has completed a `RimReplyMessage` to the sender. This function call can cause applications to block indefinitely and should be used with caution if the application doing the send is also receiving messages. For more information, refer to `RimReplyMessage`, and `RimToggleMessageReceiving`.

RimSetPID

Enables a process to change its own attributes.

```
void RimSetPID(PID * Pid)
```

Parameters

Pid	A pointer to the PID structure.
-----	---------------------------------

Description This function enables a process to change its own attributes. The name, enable/disable task switching flag, and the display icon are used during application server task-switching, and can be changed dynamically. If the Name field is NULL, the field is not modified.

For example, see `ping.c`.

RimSetReceiveFromDevice

Enables an application to specify the devices from which it wants to receive messages.

```

void RimSetReceiveFromDevice(
    DWORD Device,
    BOOL ReceiveFrom)

```

Chapter 2: System API

Parameters	Device	Defines from which device to receive or stop receiving.
	ReceiveFrom	Determines whether to start or stop receiving from the specified device.
Description	When message receiving is turned off and on by calling <code>RimToggleMessageReceiving</code> , all settings made using <code>RimSetReceiveFromDevice</code> are restored.	

RimTaskYield

Enables an application to yield control while allowing other applications to run.

```
void RimTaskYield()
```

Description Yielding control is done primarily during intensive CPU operations so that other applications can run. If there are no other applications with messages pending, `RimTaskYield` returns immediately.

For example, see `bouncer.c` and `ping.c`.

RimTerminateThread

Enables a thread to terminate itself.

```
void RimTerminateThread()
```

Description This function does not return. Calling this function has the same effect as returning from the main function of a thread.

RimToggleMessageReceiving

Enables an application to reject all incoming messages, or to accept them again after rejecting them for a period of time

```
void RimToggleMessageReceiving(BOOL ReceiveMessages)
```

Parameters `ReceiveMessages` Determines whether the application is going to receive messages or turn message receiving off.

Description This function does not change *which* messages are sent to the application, which is done with `RimSetReceiveFromDevice`. Any attempt to send a message to an application that has turned off message receiving fails and the sending call returns false.

Refer to "RimSetReceiveFromDevice" on page 21 for more information.

RimWaitForSpecificMessage

Enables an application to receive a specific message at a critical time.

```
BOOL RimWaitForSpecificMessage(
    MESSAGE * Msg,
    const MESSAGE * Compare,
    DWORD Mask)
```

Parameters

<code>Msg</code>	A pointer to the message structure that will be populated by receiving a message.
<code>compare</code>	A pointer to a message structure that will be compared with incoming messages.
<code>Mask</code>	A bit mask made up from <code>MC_DEVICE</code> , <code>MC_EVENT</code> , and <code>MC_SUBMSG</code> . This mask determines which parts of the message structure to compare.

Returns A valid mask causes the function to return true; see `RimGetMessage` for information on the values for mask. If the mask is not valid, the function returns false.

Description `RimWaitForSpecificMessage` enables an application to receive a specific message at a critical time. All other messages sent to the calling application are queued until `RimGetMessage` is called. When using this function call, care should be taken not to allow too many messages to back up in the message queue.

Refer to "RimSetReceiveFromDevice" on page 21 for more information.

Memory allocation functions

This section describes memory allocation functions, listed in alphabetical order.

RimFree	24
RimGetMaxAllocSize	24
RimMalloc	24
RimMemoryRemaining	25
RimRealloc	25

RimFree

De-allocates memory from the far heap and returns the block to the free list.

```
void RimFree(void * Block)
```

Parameters `Block` A pointer to the memory block to be freed.



Note: Attempting to free an invalid block not allocated by `RimMalloc` (`malloc`) can result in subsequent failure when allocating memory.

RimGetMaxAllocSize

Retrieves the size of the largest block of memory that can be allocated in the heap.

```
DWORD RimGetMaxAllocSize()
```

Returns The number of allocated bytes in the heap.

Description Because of fragmentation, the largest single block of memory that can be allocated is typically less than the total memory that can be allocated. For example, you might be able to allocate a 20-KB block plus a 10-KB block for a total of 30 KB, but you might not be able to allocate a single 30-KB block.

RimMalloc

Allocates memory.

```
void * RimMalloc(DWORD Size)
```

Parameters `Size` The size of the block of memory to allocate.

Returns A void pointer to a block of allocated memory or `NULL` if insufficient memory is available.

Description The memory block can be larger than the size requested due to the alignment and information required. A size of zero returns a valid pointer with a zero length size. Applications should always check the return value.

RimMemoryRemaining

Indicates the total number of bytes in unallocated blocks in the heap.

```
DWORD RimMemoryRemaining()
```

Returns The number of free bytes in the heap.

Description This function returns the total number of bytes in unallocated blocks in the heap. This is not equal to the maximum space that can be allocated using `RimMalloc`, because the free space might be fragmented into two or more free blocks. It does give an estimate of the aggregate total number of bytes which can be allocated using `RimMalloc`, but this calculation must take into account the system overhead of 10 bytes for each block, and the fact that block sizes (including overhead bytes) are always rounded up to the next multiple of 8 bytes.

RimRealloc

Reallocates memory.

```
void * RimRealloc( void * Ptr, DWORD Size)
```

Parameters

<code>Ptr</code>	A pointer to the memory block to grow or shrink.
<code>Size</code>	The desired new size of the block.

Returns A void pointer to a block of allocated memory, or NULL if a block of the requested size could not be created.

Description The block can be larger than the size requested due to the alignment and information required. A size of zero returns a valid pointer with a zero length size. Applications should always check the return value. The system can move the contents of the old block of memory if necessary to find the requested space. A NULL return simply means that a block of the requested size could not be created. The existing block of memory (as referenced by `block`) is untouched in this case. If `block` is NULL when this function is called, the function acts as `RimMalloc`.

Time functions

This section describes functions related to date and time, listed in alphabetical order.

RimGetAlarm	26
RimGetDateTime	26
RimGetIdleTime	27
RimGetTicks	27
RimKillTimer	27
RimSetAlarmClock	28
RimSetDate	28
RimSetTime	28
RimSetTimer	29
RimSleep	30

RimGetAlarm

Finds the earliest enabled alarm set by any application.

```
TASK RimGetAlarm(TIME * Time)
```

Parameters `Time` A pointer to a `time` structure to be completed with the settings for the next alarm.

Returns The handle of the task that has set the earliest future alarm date and time using `RimSetAlarmClock`. If no alarms were enabled, `RimGetAlarm` returns 0.

Description This function is used to find the earliest enabled alarm set by any application.

RimGetDateTime

Gets the date and time from the real-time clock.

```
void RimGetDateTime(TIME * Time)
```

Parameters `Time` A pointer to a `time` structure.

Description This function gets the date and time from the real-time clock. The time is always in 24-hour format.

The earliest date and time that can be returned by `RimGetDateTime` is 01/01/1998 00:00:00. The latest date and time which can be returned by `RimGetDateTime` in the embedded system is 12/31/2090 23:59:59. The Windows simulator, however, uses `mktime` which can only handle a date from January 1, 1970, to midnight, February 5, 2036. Therefore, `RimGetDateTime` in the Windows simulator cannot return any date later than February, 5, 2036.

For example, see `realtime.c`.

RimGetIdleTime

Retrieves the number of seconds since the previous key or trackwheel event.

```
long RimGetIdleTime()
```

Returns The number of seconds since the last key or trackwheel event. A value of 300, for example, would indicate 5 minutes.

Description The value returned by this function can be used to decide when to enable a screen saver, or to implement a security timeout.

RimGetTicks

Tracks the amount of time since the handheld was turned on.

```
long RimGetTicks()
```

Returns The number of time ticks, in 10 millisecond increments, since the handheld was turned on. A value of 5000, for example, would indicate 50 seconds.

Description The value returned by this function can be used to reference absolute timers (`RimSetTimer`). By obtaining this value, you can avoid the potential drift in periodic timers if the system does not process the timers fast enough.

For example, see `regmess.c`.

RimKillTimer

Cancels a timer that was previously set by the application.

```
void RimKillTimer(DWORD TimerID)
```

Parameters `TimerID` The identifier of the time to be cancelled.

Description This function cancels a timer that was previously set by the application. The `TimerID` should match the value passed to a previous call to `RimSetTimer`. The caller will not receive more timer device messages with the specified `TimerID` after the call returns, even if the timer message was already in the receiving task's message queue before `RimKillTimer` was called.

For examples, see `bouncer.c` and `regmess.c`.

RimSetAlarmClock

Sets the time for an application to be notified when that date and time has expired.

```
BOOL RimSetAlarmClock( TIME * Time, BOOL Enable)
```

Parameters

<code>Time</code>	A pointer to a <code>TIME</code> structure.
<code>Enable</code>	Disables or enables the alarm clock.

Returns True if the alarm was set or false if there was an error.

Description This function sets the time for an application to be notified when that date and time has expired. When the time has expired, the `alarm_expired` event is sent to the task. To enable the alarm clock, use true for the `Enable` parameter. To disable the alarm clock, use false for the `enable` parameter.



Note: Each application can set its own alarm time.

For example, see `realtime.c`.

RimSetDate

Sets the date on the real-time clock.

```
BOOL RimSetDate(TIME * Time)
```

Parameters

<code>Time</code>	A pointer to a <code>TIME</code> structure.
-------------------	---

Returns True if the date was set or false if the date was incorrectly formatted.
For example, see `realtime.c`.

RimSetTime

Sets the time on the device real-time clock.

```
BOOL RimSetTime(const TIME * Time)
```

Parameters

<code>Time</code>	A pointer to a <code>TIME</code> structure.
-------------------	---

Returns True if the time was set or false if the time was incorrectly formatted.

Description This function sets the time on the device real-time clock. The programmed time must be in 24-hour format.

The `TIME` structure includes a byte which carries time zone information. The value stored in the `TIME` structure (excluding the time zone byte) is the local time on the device. To convert to GMT, the offset given by the time zone must be subtracted from the local time. Care must be taken to manage transitions into differing days, months, and years.

For example, see `realtime.c`.

RimSetTimer

Sets a timer for the application.

```
BOOL RimSetTimer(
    DWORD TimerID,
    DWORD Time,
    DWORD Type)
```

Parameters	<code>TimerID</code>	The identifier of the timer to be set.
	<code>Time</code>	The amount of time until the timer expires, and the time between subsequent timer expirations for periodic timers, specified in 1/100 second increments.
	<code>Type</code>	This parameter can be one of three values.

Value	Description
<code>TIMER_ONE_SHOT</code>	The timer will expire once, in the time specified by <code>time</code> .
<code>TIMER_PERIODIC</code>	The timer will expire with a period specified by <code>time</code> .
<code>TIMER_ABSOLUTE</code>	The timer will expire when the absolute tick count reaches the specified value. The absolute tick count can be obtained by <code>RimGetTicks()</code> .

Returns True if the timer was set successfully or false otherwise.

Description This function sets a timer for the application. If a timer with the same `timerID` was previously set by that application, it is cancelled. The application receives a message from the Timer device after the specified period of time. For periodic timers, the messages arrive after each time period beyond that.

The `timerID` is an arbitrary identifier that is assigned by, and is local to, the calling task. There is no correlation between `TimerIDs` and the internal identifiers of the global timer pool.

When the timer expires, the application receives a `DEVICE_TIMER` event.

For examples, see `bouncer.c`, `clock.c`, and `regmess.c`.

RimSleep

Enables an application to stop running for a specified period of time.

```
void RimSleep(DWORD Ticks)
```

Parameters Ticks Defines the number of 10 millisecond increments that the task is to sleep.

Description The `RimSleep` function enables an application to stop running for a specified period of time. No messages wake the application so care must be taken to ensure that message queues do not over flow.

Refer to "RimToggleMessageReceiving" on page 23 for more information.

User notification functions

This section describes user notification functions, listed in alphabetical order.

<code>RimAlertNotify</code>	30
<code>RimGetAlertConfiguration</code>	31
<code>RimGetNumberOfTunes</code>	31
<code>RimGetTuneName</code>	31
<code>RimSetAlertConfiguration</code>	32
<code>RimSpeakerBeep</code>	32
<code>RimTestAlert</code>	32

RimAlertNotify

Notifies the user of an event such as an incoming message, using audible tones, vibrations, or both.

```
void RimAlertNotify(  
    int Notify,  
    int MaxRepetitions)
```

Parameters `Notify` Set to one of `NO_NOTIFY`, `KILL_NOTIFY`, or `TONE_SEQUENCE`.

`MaxRepetitions` This parameter can be used to limit how many times the tone sequence is played. Set `MaxRepetitions` to `-1` to use the system default number of repetitions.

Description This function is used to notify the user of an event, such as an incoming message, using audible tones, vibrations, or both. When generating tones, it is possible to specify one of the six built-in tone sequences. The alert notify can be cancelled when `notify` is set to `kill_notify`. The alert notify is automatically cancelled by the operating system if the user presses a key while the notification is happening.

For example, see `bouncer.c`.

RimGetAlertConfiguration

Determine the alert settings that are currently in effect.

```
void RimGetAlertConfiguration(AlertConfiguration * AlertConfig )
```

Parameters `AlertConfig` A pointer to an `AlertConfiguration` structure to be filled with the current configuration of `RimAlertNotify`. Refer to "AlertConfiguration" on page 7 for more information.

Description This function can be used by a user options application to determine the settings currently in effect, in preparation for calling `RimSetAlertConfiguration`.

RimGetNumberOfTunes

Retrieves the total number of tunes (predefined and resource).

```
unsigned int RimGetNumberOfTunes()
```

Returns Total number of tunes available.

Description Valid indices for tunes are 0 to `RimGetNumberOfTunes()`. (Zero is silence.) By default there are 6 tunes installed (`TONE_SEQUENCE_1` through `TONE_SEQUENCE_6`). If resource tunes were installed in a resource DLL, the number returned is 6 + number of tunes in the resource DLL. The first resource tune installed starts at 7.

Example

```
int num tunes;
// Obtain the total number of tunes installed
numTunes = RimGetNumberOfTunes ();
```

RimGetTuneName

Retrieves the name of a tune.

```
int RimGetTuneName(
    unsigned int tuneIndex,
    const char ** tuneName)
```

Parameters `tuneIndex` Index of the tune in the range 0..`RimGetNumberOfTunes()` -1.

`tuneName` A pointer to a string that points to the tune name after the function is executed.

Returns `TUNE_OK` if `tuneIndex` is within the range of currently available tunes (see `RimGetNumberOfTunes`). If the index is out of range, the function returns `BAD_TUNE_INDEX`.

Description To play a tune, use `RimAlertNotify()`.

Example

```
const char * tuneName;
if (RimGetTuneName (3, &tuneName ) == TUNE_OK) (
    //Use tune Name
)
```

RimSetAlertConfiguration

Specifies global changes to the behavior of RimAlertNotify.

```
void RimSetAlertConfiguration(const AlertConfiguration * AlertConfig)
```

Parameters AlertConfig A structure which sets the various options to be used when RimAlertNotify is called. Refer to "AlertConfiguration" on page 7 for more information.

Description This function can be used by a user options application to make global changes to the behavior of RimAlertNotify.

RimSpeakerBeep

Generates speaker tone.

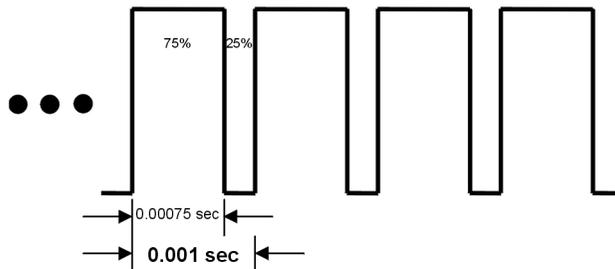
```
void RimSpeakerBeep(  
    int Frequency,  
    int Duration,  
    int Duty,  
    int Volume)
```

Parameters

Frequency	The frequency of the tone from 10 Hz to 10,200 Hz.
Duration	The duration of the beep in milliseconds from 0 sec to 2.5 sec in 10 millisecond.
Duty	The percentage ratio of the on/off frequency pulse from 1 to 99 percent.
Volume	The beep volume, from 1 percent (softest) to 100 percent (loudest).

Description This function enables the generation of different tones. The duty cycle must be in the range of 1 to 99. The default is 50.

As an example, the diagram below shows a 1000 Hz tone with a duty cycle of 75%.



RimTestAlert

Demonstrates RimAlertNotify configurations to the user.

```
void RimTestAlert()
```

```
int notify,
AlertConfiguration * TrialConfig)
```

Parameters

notify	Set this parameter to one of: NO_NOTIFY, KILL_NOTIFY, TONE_SEQUENCE_1 through TONE_SEQUENCE_6.
TrialConfig	The trial AlertConfiguration structure.

Description This function is used to demonstrate RimAlertNotify configurations to the user. When the user accepts the settings, they can be set permanently by calling RimSetAlertConfiguration.

Password functions

This section describes functions related to the handheld password. Functions are listed in alphabetical order.

RimSetPassword	33
RimVerifyPassword	33
RimPasswordFailureCount	34

RimSetPassword

Sets the handheld password.

```
BOOL RimSetPassword(
    const char *oldPasswordHash,
    const char *newPasswordHash)
```

Parameters

oldPasswordHash	The hash of the current handheld password.
newPasswordHash	The hash of the new password to set.

Returns True if the password is successfully changed; otherwise false.

RimVerifyPassword

Verifies the current handheld password.

```
BOOL RimVerifyPassword(const char *passwordHash)
```

Parameters

PasswordHash	The hash of the current handheld password.
--------------	--

Returns True if the value of PasswordHash is correct; false if the function call fails or if the value of PasswordHash is incorrect.

RimPasswordFailureCount

Retrieves the number of times that an incorrect password has been entered on the handheld, since the last time the user entered the correct password.

DWORD RimGetPasswordFailureCount()

Returns True if the value of PasswordHash is correct; false if the function call fails or if the value of PasswordHash is incorrect.

Description Calling RimSetPassword or RimVerifyPassword with the incorrect hash increments the count; calling either function with the correct hash resets the count to zero.

Miscellaneous functions

This section describes miscellaneous system functions, listed in alphabetical order.

RimCatastrophicFailure	35
RimConfigureLEDs	35
RimDebugPrintf	36
RimGetBatteryLevel	36
RimGetBatteryStatus	37
RimGetDeviceInfo	37
RimGetLanguage	38
RimGetLoadedAppInfo	38
RimGetMessageRaw	39
RimGetOSversion	40
RimGetSetOfLanguages	41
RimHolsterStatus	41
RimInitiateReset	42
RimPowerDownHandled	42
RimRegisterForPowerDown	42
RimRequestFullPowerOff	42
RimRequestPowerOff	43
RimRequestStorageMode	43
RimSetLanguage	43
RimSetLed	44
RimSprintf	44
RimStackUsage	46
RimVsprintf	46

RimCatastrophicFailure

Handles unrecoverable application errors.

```
void RimCatastrophicFailure(char * FailureMessage)
```

Parameters FailureMessage A pointer to a string to be displayed on the handheld.

Description This function is used to handle unrecoverable application errors. RimCatastrophicFailure displays that an unrecoverable error has occurred on the handheld, and prompts the user to push a key to cause the handheld to be reset. This mechanism is used to handle error conditions that cannot be recovered without resetting the system.

For example, see regmess.c.

RimConfigureLEDs

Configures LED lighting (duration, frequency, and brightness).

```
void RimConfigureLEDs(
    int LED_On_Time,
    int LED_Off_Time,
    int DutyCycle)
```

Parameters

LED_On_Time	Length of time in milliseconds that the LED should remain on (rounded to the nearest 8 ms).
LED_Off_Time	Length of time in milliseconds that the LED should remain off (rounded to the nearest 64 ms).
DutyCycle	Sets the brightness of the LED while it is on. It is a percentage of the length of LED_On_Time that the LED will be on. Valid values are: <ul style="list-style-type: none"> • LED_DUTY_12 for 12% • LED_DUTY_25 for 25% • LED_DUTY_50 for 50% • LED_DUTY_100 for 100%

Description This call is only available on the palm-sized RIM Wireless Handheld. For example, see RimSetLed.

RimDebugPrintf

Prints formatted text to debug stream.

```
void RimDebugPrintf(const char * String, ...)
```

Parameters String A format control string, as used in the standard library function printf.

Description The RimDebugPrintf function formats and prints a series of characters and values to the debug output stream, which can be displayed in the Output window of Developer Studio. On the handheld, RimDebugPrintf has no effect. If arguments follow the format string, the format string must contain specifications that determine the output format for the arguments.

RimGetBatteryLevel

Indicates the percentage of battery life remaining.

```
DWORD RimGetBatteryLevel()
```

Returns The charge remaining in the battery, as a percentage.

Description The charge remaining in the battery is a good indication of how much operating time is left.

RimGetBatteryStatus

Retrieves the current state of the battery or batteries.

DWORD RimGetBatteryStatus()

Returns A bitmask of a combination of the following flags:

Flag	Description
BSTAT_DEAD	Battery is at 0%
BSTAT_TOO_COLD	Battery is too cold
BSTAT_TOO_HOT	Battery is too hot
BSTAT_LOW™	Battery is low
BSTAT_NONE	No AA battery (RIM 850™ and RIM 950™)
BSTAT_REVERSED	AA battery is inserted backwards (RIM 850™ and RIM 950™)
BSTAT_NO_TURN_ON	Device cannot turn on
BSTAT_NO_RADIO	Radio cannot send
BSTAT_CHARGING	Battery is charging

Description This function is usually called upon receiving a BATTERY_UPDATE event from the system. A return value of 0 means that the battery is fine.

RimGetDeviceInfo

Retrieves information about the handheld.

VOID RimGetDeviceInfo(DEVICE_INFO * Info)

Parameters Info A pointer to a DEVICE_INFO structure that will receive the information about the device.

Description Use this function to determine at runtime the type of device and network the application is running on. See page 9 for information on the system structures, such as the DEVICE_INFO structure.

RimGetLanguage

Retrieves language code.

```
int RimGetLanguage()
```

Returns The language code as last set by `RimSetLanguage`.

Description This function could be used by applications to customize country- or language-dependent behavior.

RimGetLoadedAppInfo

Retrieves information about each application on the handheld.

```
void RimGetLoadedAppInfo(LoadedAppInfo * Info)
```

Parameters Info A pointer to a `LoadedAppInfo` structure that stores application information.

Description This function enables an application to determine which applications are loaded on the handheld. This function does not work on the simulator.

Use the function as follows:

- Define a `LoadedAppInfo` structure with the `LoadTableAddress` field set to `NULL`.
- Call `RimGetLoadedAppInfo` with a pointer to this structure.
If `LoadTableAddress` is not `NULL`, the `Name`, `VersionLS`, `VersionMS`, `Checksum`, `TimeStamp` and `Size` fields are valid.
- You can call the function again, without changing `LoadTableAddress`, to get information about the next DLL.
If `LoadTableAddress` is `NULL`, the end of the list has been reached.

Structure The `LoadedAppInfo` structure is as follows:

```
typedef struct {  
    void *LoadTableAddress;  
    DWORD VersionLS;  
    DWORD VersionMS;  
    DWORD Checksum;  
    DWORD TimeDateStamp;  
    DWORD Size;  
    DWORD Spare2;  
    DWORD Spare3;  
    char Name[40];  
} LoadedAppInfo;
```

The following table describes each field.

Field	Description
LoadTableAddress	Address of the load table for the application; used internally by the operating system.
VersionLS	The Least Significant DWORD of the version number, as supplied by the boot ROM.
VersionMS	The Most Significant DWORD of the version number, as supplied by the boot ROM.
Checksum	The Checksum of the module, which enables a comparison of an application loaded on the handheld with a DLL on the desktop.
TimeDateStamp	The date and time when the DLL was created.
Size	The total amount of flash used by this application.
Spare2	Reserved for future use.
Spare3	Reserved for future use.
Name [40]	The name of the application, such as SerialDbAccess.dll.

RimGetMessageRaw

Retrieves the next message from its message queue without triggering any registered callbacks.

```
TASK RimGetMessageRaw(MESSAGE * Msg)
```

Parameters `Msg` A pointer to a MESSAGE structure that receives the message.

Returns The task handle of the task that sent or posted the received message received.

Description The functional semantics of `RimGetMessageRaw` are identical to that of `RimGetMessage`. The difference revolves around the handling of registered callbacks (see `RimRegisterMessageCallback`). This function enables the calling application to obtain the next message from its message queue without triggering any registered callbacks. `RimGetMessageRaw` can be called from within a registered callback function.

RimGetOSversion

Compares the version of the SDK that an application was built with to the version of the operating system that the application is running on.

DWORD RimGetOSversion()

Returns The operating system version. The version is as follows:
Version.Revision.Release.Build

These four values are packed into the DWORD value as follows:

Value	Description
Version	Bits 31-24 (most significant bits)
Revision	Bits 23-16
Release	Bits 15-8
Build	Bits 7-0

Description The operating system version, at build time, is also available in the constant OS_API_VERSION. This function can be used to compare the version of the SDK that an application was built with to the version of the operating system on which the application is running.

RimGetSetOfLanguages

Retrieves languages included in the localization DLL.

```
BOOL RimGetSetOfLanguages(AVAILABLE_LANGUAGES **LangList )
```

- Parameters** `LangList` A pointer to an AVAILABLE_LANGUAGES structure that receives the message.
- Returns** True if the `LangList` pointer is set successfully; false otherwise.
- Description** This function sets a pointer to a structure that describes the languages included in the localization DLL.
- Structures** The AVAILABLE_LANGUAGES structure is as follows:

```
typedef struct {
    const unsigned int  VERSION;
    const unsigned int  numLangs;
    const LANGUAGE_DEF * const langs[];
} AVAILABLE_LANGUAGES;
```

The LANGUAGE_DEF structure is as follows:

```
typedef struct {
    const char * const langName;
    const unsigned int intLangId;
    const char * const langString[];
} LANGUAGE_DEF;
```

Refer to "RimGetLanguage" on page 38 for more information.

RimHolsterStatus

Retrieves status of handheld in or out of holster.

```
int RimHolsterStatus()
```

- Returns** IN_HOLSTER if the handheld is in its holster, or OUT_OF_HOLSTER if it is out of the holster.
- Description** If the status changes while the handheld is not in POWER_OFF mode, all tasks are notified of this by a DEVICE_HOLSTER message. This function can be used to detect the state at POWER_UP if required.

RimInitiateReset

Resets the handheld.

```
void RimInitiateReset()
```

Description This function causes the handheld to reset as if it was powering up.

RimPowerDownHandled

Enables the device to power down.

```
void RimPowerDownHandled()
```

Description Applications that call `RimRegisterForPowerDown` receive a `POWER_OFF` system event during power down. After processing this message, `RimPowerDownHandled` should be called to complete the power down process.

RimRegisterForPowerDown

Prevents the device from powering down until the application calls `RimPowerDownHandled`.

```
BOOL RimRegisterForPowerDown()
```

Returns True if successful; false otherwise.

Description Applications that call `RimRegisterForPowerDown` receive a `POWER_OFF` system event during power down. After processing this message, call `RimPowerDownHandled` to complete the power down process.

RimRequestFullPowerOff

Sends a request to the handheld to turn power off.

```
BOOL RimRequestFullPowerOff()
```

Returns True if the power off sequence was started. In this case, a subsequent `RimGetMessage` receives a `POWER_OFF` event.

False if the handheld cannot power down because the COM port is in use by an application.

Description This function posts a `POWER_OFF_REQUEST` message to the System Task, which then turns the radio off, blanks the LCD and sends a `POWER_OFF` notification to all application tasks.

When the handheld is turned off by `RimRequestFullPowerOff`, an alarm does not turn the handheld back on.

Refer to "RimRequestPowerOff" on page 43 for more information.

RimRequestPowerOff

Sends a request to the handheld to turn power off.

`BOOL RimRequestPowerOff()`

Returns True if the power off sequence was started. In this case, a subsequent `RimGetMessage` receives a `POWER_OFF` event. Otherwise, the function returns false if power down is not possible as the COM port is owned by an application. This prevents events such as auto shut-off from spontaneously shutting the handheld down while synchronizing or while connected to the RIM Wireless Handheld simulator.

Description This function posts a `POWER_OFF_REQUEST` message to the System Task, which then turns the radio off, blanks the LCD, and sends a `POWER_OFF` notification to all application tasks.

When the handheld is turned off by `RimRequestPowerOff`, an alarm turns the handheld back on. Refer to "RimRequestFullPowerOff" on page 42 for more information.

For example, see `bouncer.c`.

RimRequestStorageMode

Sends a request to put the handheld into a storage state to preserve the battery.

`BOOL RimRequestStorageMode()`

Returns Returns true if the handheld is placed into storage mode; otherwise false.

Description This function electronically disconnects the Lithium battery to prevent it from draining over a long period of inactivity. The real-time clock will probably drift during the time the handheld is in this state.

The only way to power up the device after this call is to press the reset button on the back of the device or to place it in a charging cradle.

This function is only available on the palm-sized Wireless Handheld.

RimSetLanguage

Sets language code.

`void RimSetLanguage(unsigned int Language)`

Parameters `Language` The language code, depending on which language DLLs are loaded.

Description This function provides a method for customizing country- or language-dependent behavior.

RimSetLed

Sets the state of either the coverage LED or the message LED.

```
void RimSetLed(  
    int LedNumber,  
    int LedState)
```

Parameters

LedNumber	The LED to be set, either LED_COVERAGE for the coverage LED or LED_MESSAGE for the message LED.
LedState	One of LED_OFF, LED_ON, or LED_BLINK.

Description This function is only available on the palm-sized Wireless Handheld.

RimSprintf

Formats text into a buffer.

```
int RimSprintf(  
    char * Str,  
    int MaxLen,  
    char * Fmt,  
    ... )
```

Parameters

Str	The supplied buffer where the formatted string is to be placed.
MaxLen	The maximum number of characters that can be placed into the buffer.
Fmt	The format string used to determine the format of the data.
...	Following the format string is the list of parameters as necessary to supply information to the format string.

Returns The number of characters placed into Buffer, excluding terminating NULL character. -1 is returned if there was insufficient space, in which case Maxsize characters have been placed into the buffer.



Note: If -1 is returned the NULL termination has not been added to the buffer.

Description This function is similar to the standard C function `sprintf`, except with a buffer size parameter added. Field specifications are in the form:

`% [-] [0] [width] [.precision] [1] format`

(Optional parts are in []. Do not include the [] characters)

Format specifier	Description
-	This left-justifies the field; it is right-justified otherwise.
0	Pad with '0' to the width of the field, only if it is right justified. The default is to use spaces.
width	Specify the minimum width of the formatted data for this field.
precision	For numbers, specify the minimum number of digits to display. For strings, specify the maximum number of characters to display.
l	Indicate that the number to be formatted is a long integer value.
format	<p>a format type, one of:</p> <ul style="list-style-type: none"> d Data is an integer, displayed in decimal format. i Data is an integer, displayed in decimal format. b Data is an integer, displayed in binary format. o Data is an integer, displayed in octal format. u Data is an unsigned integer, displayed in decimal format. x Data is an integer, displayed in hexadecimal format, using lowercase letters. X Data is an integer, displayed in hexadecimal format, using uppercase letters. s Data is a pointer to a string; if a <code>precision</code> is not specified, the string must be NULL-terminated. c Data is a character.

For examples, see `hello.c`, `ping.c`, `realtime.c`, and `regmess.c`.

RimStackUsage

Helps choose an appropriate value for the task stack size.

```
DWORD RimStackUsage()
```

Returns The maximum stack size used by the current task since the previous call to `RimStackUsage`. This number includes the stack space required by `RimStackUsage` itself.

Description This function can be used to help choose an appropriate value for the task stack size. The task stack size is specified as the constant `appstacksize` for `PagerMain` functions or as an parameter to `RimCreateThread` for other functions.



Note: Since this function checks for a high-water mark, the results returned on the first call by any task might not be meaningful.

RimVsprintf

Formats text into a buffer.

```
int RimVsprintf(  
    char * Buf,  
    int Maxsize,  
    char * Fmt,  
    va_list Argp)
```

Parameters	Buf	The supplied buffer where the formatted string is to be placed.
	Maxsize	The maximum number of characters that can be placed in the buffer.
	Fmt	The format string used to determine the format of the data.
	Argp	A pointer to a variable parameter list. The list of parameters is provided to supply information to the format string.

Description This function behaves similarly to the standard C library function `vsprintf` and identically to `RimSprintf`. Refer to "RimSprintf" on page 44 for more information.

Chapter 3

File System API

The File System API provides access to the persistent storage on the handheld.

If your application requires additional functionality, you can use the Database API. Refer to the *Database API Reference Guide* for more information.

Structures

The following structures are used by the database / file system APIs.

FileInfoType	47
FileSysInfoType	48

FileInfoType

The FileInfoType structure (used by DbFileInfo) looks like this:

```
typedef struct {
    unsigned Pos;
    unsigned Length;
    HandleType Db;
} FileInfoType;
```

Field	Description
Pos	This field contains the current read/write position in the file (zero-based). The range of Pos is from 0 to Length, inclusive.
Length	This field contains the length of the file, in bytes. Immediately after DbFileOpen, the Length is equal to the database length, i.e. to the total size of all the database records. The Length can be increased by performing DbFileWrite or DbFileSeek.
Db	Database handle of the file

FileSysInfoType

The FileSysInfoType structure (used by DbFileSysInfo) looks like this:

```
typedef struct {
    void * Disk;
    unsigned NumOfBlock;
    unsigned BlockSize;
    unsigned NumOfClean;
} FileSysInfoType;
```

Field	Description
Disk	Starting address of the file system's data
NumOfBlock	Number of flash memory blocks in the file system
BlockSize	Size of each flash memory block in the file system
NumOfClean	Number of cleanups that have taken place since the handheld was turned on

Functions

The following file system functions are listed alphabetically.

DbAddOrphan	49
DbAddRec	49
DbAndRec	51
DbDelete	52
DbDeleteRec	52
DbFileClose	53
DbFileInfo	54
DbFileOpen	54
DbFileRead	55
DbFileSeek	55
DbFileSysInfo	56
DbFileWrite	57
DbFindNext	58
DbFirstRec	59
DbFreeRec	59
DbFreeSpace	59
DbGetHandle	60
DbMaxHandles	61
DbMaxNewRecSize	61
DbName	62
DbNextRec	62
DbPointTable	63

DbPointTableEdition 63
 DbRecSize 63
 DbReplaceOrphan 64
 DbReplaceRec 64
 DbSecure 65
 DbSize 65

DbAddOrphan

Adds an orphan record to a database.

```
STATUS DbAddOrphan(
    HandleType db,
    HandleType orphan)
```

Parameters

db A database handle.

orphan The handle to the orphan record that is to be appended to the database.

Returns One of the following return codes is returned:

```
DB_OK
DB_ERR_BAD_HANDLE
DB_ERR_NOT_DIR
DB_ERR_NOT_ORPHAN
```

Description This function is atomic. The record is either appended to the database or the database is unchanged. No intermediate state is visible. Data consistency is preserved if a system error occurs when the function is executing.

This function cannot be used to move a record from one database to another. To do this, create a new record, copy the contents of the old record (using DbAddRec) and delete the old record.

An orphan record is a database record that has not yet been assigned to any database. This is useful for allocating temporary data in the file system.

DbAddRec

Creates a new record (database or orphan).

```
HandleType DbAddRec(
    HandleType db,
    unsigned size,
    const void * data)
```

Chapter 3: File System API

Parameters	db	The handle to the database to which the record is to be appended; a negative value indicates that an orphan record should be created.
	size	The size of the record, in bytes. It must be between 0 and $2^{16}-2$, inclusive; in addition, the maximum possible record size is limited due to flash memory fragmentation. Refer to "DbMaxNewRecSize" on page 61 for more information.
	data	The pointer to the data to be written into the new record; if it is NULL, the data is assumed to have all bits set to 1 (this is convenient when used with the DbAndRec function).

Returns A non-negative handle to the newly-created record, or one of the following negative error codes:

DB_ERR_BAD_SIZE
DB_ERR_NO_HANDLE
DB_ERR_NO_SPACE
DB_ERR_BAD_HANDLE
DB_ERR_NOT_DIR



Warning: The handle returned by this function persists only for the lifetime of the database on the particular system; it is intended for temporary database identification only. Do not store record or database handles in the permanent data because the data cannot be meaningfully copied to another system.

Description This function is atomic. The record is either created completely or the database is unchanged. No intermediate state is visible. Data consistency is preserved if a system error occurs when the function is executing.

It is valid for data to point to portions of the file system, for example, through the memory mapped file access mechanism; however, the entire operation is slightly slower. In order to write to the flash memory, the memory must be configured to preclude simultaneously reading it. As a result, the function must copy from flash memory to flash memory through RAM.

An orphan record is a database record that has not yet been assigned to any database. This is useful for allocating temporary data in the file system. It is also useful if creating a record and adding it to a database should be kept separate. Note that orphan database records are regular database records except that they are not associated with any file. Orphan records are automatically deleted after a reset; it is not possible to get the handle to an orphaned record.

DbAndRec

Logical AND of data to an existing record.

```
STATUS DbAndRec(
    HandleType rec,
    const void * mask,
    unsigned size,
    unsigned offSet)
```

Parameters	<code>rec</code>	The record to be modified.
	<code>mask</code>	A pointer to the data mask to be AND-ed to the current contents of the record. If it is NULL, the data mask is assumed to have all bits cleared to zero.
	<code>size</code>	The data mask size in bytes.
	<code>offSet</code>	The offset of the area to be AND-ed to from the beginning of record.



Note: As a pre-condition, `offSet+size` must be less than or equal to the record size.

Returns One of the following return codes:

```
DB_OK
DB_ERR_BOUNDS
DB_ERR_BAD_HANDLE
DB_ERR_IS_DIR
```

Description Changes made by the `DbAndRec` function are only partially verified. It is verified that bits are set to 0 as specified by the data mask, but not that all remaining bits are unchanged. In this case, you must ensure that the data is correctly interpreted.

This function is not atomic. If a system error occurs when the function is executing, the record can be left in a partially updated state. The update order is not specified. You must ensure that the data is correctly interpreted.

`DbAndRec` should only be used with non-critical, transient data. `DbAndRec` is not atomic and changes are not verified completely. The advantage of `DbAndRec` is that it performs record modification in place, which uses less RAM and is faster than copying the record, modifying the copy and then calling `DbReplaceRec` to make the changes permanent.

DbDelete

Deletes a database.

```
STATUS DbDelete(char * fileName)
```

Parameters `fileName` A pointer to the name of the database to be deleted from the file system.

Returns One of the following return codes is returned:

```
DB_OK  
DB_ERR_FILE_OPEN  
DB_ERR_BAD_NAME  
DB_ERR_NOT_EXIST
```

Description When a database is deleted, its records are also deleted. In addition, the handles associated with the database (that is, the database handle and all the record handles) are immediately made available for reuse.

The system cannot detect erroneous use of outdated handles. Applications should not maintain references to deleted items. Immediately after an item is deleted, the corresponding entry in the handle mapping table is set to NULL, so that applications are more likely to detect the deleted item. Due to the limited number of handles, these handles are eventually reused for another purpose.

The following sample code demonstrates the use of this function:

```
char * name = "timestamps";  
ForgetHandles (name);  
DbDelete (name);
```

DbDeleteRec

Deletes a record.

```
STATUS DbDeleteRec(  
    HandleType db,  
    HandleType rec)
```

Parameters `db` The handle to the database containing the record; for an orphan record, this parameter is a negative value.

`rec` The record to be deleted.

Returns One of the following return codes is returned:

```
DB_OK  
DB_ERR_BAD_HANDLE  
DB_ERR_NOT_DIR  
DB_ERR_IS_DIR
```

Description This function is atomic. The record is either completely deleted from the database or the database is unchanged. No intermediate state is visible. Data consistency is preserved if a system error occurs when the function is executing.

Deleting a record frees the record handle for immediate reuse.

Generally, the cost of a delete is, on average, half the cost of the general purpose update. It is more efficient to let `DbDelete` delete all of the records of a database at once than to call `DbDeleteRec` repeatedly to delete them one at a time.

DbFileClose

Closes a file.

```
STATUS DbFileClose(
    FileType file,
    BOOL trim)
```

Parameters

<code>file</code>	A file number.
<code>trim</code>	<p>This parameter has a value of true or false:</p> <ul style="list-style-type: none"> • true: The partially used last record is trimmed. This makes the database length equal to the file <code>DbFileClose</code> immediately before <code>DbFileClose</code>. • false: The partially used last record is not altered, and the database length might be greater than the file <code>Length</code> immediately before <code>DbFileClose</code>. <p>This parameter is important if the database was extended by <code>DbFileSeek</code> or <code>DbFileWrite</code>. In this case the last appended record (of size <code>newRecSize</code>) is only partially written with data and the rest of the record is filled with bytes consisting of all ones.</p>

Returns One of the following return codes:

```
DB_OK
DB_ERR_FILE_CLOSED
DB_ERR_BAD_FILE
```

Description The database updates performed by this function or by previous calls to `DbFileSeek` or `DbFileWrite` are not atomic. They are not guaranteed to be permanent until the `DbFileClose` returns `DB_OK`. If an error occurs when the function is executing, the database can be left in a partially updated state. The update order is not specified. You must ensure that the data is correctly interpreted.

DbFileInfo

Retrieves information about a file.

```
BOOL DbFileInfo(  
    FileType file,  
    FileInfoType * fileInfo)
```

Parameters

file	A file number.
fileInfo	A pointer to a buffer into which the file information should be written.

Returns True if the file exists; false otherwise.

Description For a description of the FileInfoType structure, see “Structures” on page 47.

DbFileOpen

Opens a file.

```
FileType DbFileOpen(  
    HandleType db,  
    unsigned newRecSize)
```

Parameters

db	The handle to the database which is to be opened as a streamed file.
newRecSize	The size of new records to be written to the database whenever writing to the file requires extending the underlying database. This value must be between 1 and $2^{16}-2$, inclusive. At any particular instant, the maximum possible record size is additionally limited due to flash memory fragmentation. Refer to “DbMaxNewRecSize” on page 61 for more information.

Returns A non-negative file number (as distinct from a database or record handles), or one of the following negative error codes:

```
DB_ERR_BAD_HANDLE  
DB_ERR_BAD_SIZE  
DB_ERR_FILE_OPEN  
DB_ERR_NOT_DIR  
DB_ERR_PRIV_HANDLE  
DB_ERR_NO_FILE
```

Description Call DbGetHandle before calling DbFileOpen to obtain the database handle and create the database if it does not exist.

The database used to implement the file can contain previously created data arranged in a collection of records of arbitrary length. If the file mechanism is used to overwrite some of this data, the file view of the database preserves the previously existing record structure. If the file is extended beyond the end of the database, then all new records are of length newRecSize. For more information, refer to DbGetHandle.

DbFileRead

Reads bytes from file.

```
int DbFileRead(
    FileType file,
    void * data,
    unsigned size)
```

Parameters

file	A file number.
data	A pointer to the destination buffer.
size	The number of bytes to be read from the file.

Returns The non-negative number of bytes actually read, or one of the following negative error codes:

```
DB_ERR_BAD_HANDLE
DB_ERR_FILE_CLOSED
```

Description Reading is performed sequentially from the current file position and is independent of the record structure of the underlying database.

The data is read up to the end of the file or until the number of bytes specified by the size parameter have been read. It is not an error to request to read past the end of the file. If the current position prior to `DbFileRead` was at the end of the file, no bytes are read, and the return value is zero.

DbFileSeek

Sets the current position in a file, possibly extending the file.

```
STATUS DbFileSeek(
    FileType file,
    unsigned pos)
```

Parameters

file	A file number.
pos	The desired zero-based current position.

Returns One of the following return codes is returned:

```
DB_OK
DB_ERR_BAD_FILE
DB_ERR_FILE_CLOSED
DB_ERR_NO_HANDLE
DB_ERR_NO_SPACE
```

Chapter 3: File System API

Description If `pos` is greater than `Length`, the file is extended, with `pos - Length` bytes consisting of all ones. Both the current position and `Length` is made equal to the supplied `pos`. In this case the appended records have the size specified by the `newRecSize` parameter of `DbFileOpen`. The last appended record might be only partially used, with the consequence of `Length` being smaller than the database length. This is important because `Length` is not stored permanently, unless `DbFileClose` is performed with `trim` equal to `true`.

The file extension is subject to having enough free space at the moment for all the new records required, each of a size `newRecSize`, that is, the system does not attempt to create a smaller record if there is not enough space. Instead, `DB_ERR_NO_SPACE` is returned.

The file extension process is not atomic, and it is not guaranteed to be permanent until the subsequent `DbFileClose` returns `DB_OK`. If a system error occurs before `DB_OK` is returned, the database might be left in a partially updated state. In this case, you must ensure that the data is correctly interpreted.

Seeking forward is significantly faster than seeking backward.

DbFileSysInfo

Retrieves information about the file system.

```
void DbFileSysInfo(FileSysInfoType * FileSysInfo)
```

Parameters `FileSysInfo` A pointer to the buffer to which the file system information should be written.

Description For a description of the `FileSysInfoType` structure, see “Structures” on page 47.

The following sample code demonstrates the use of this function:

```
FileSysInfoType FileSysInfo;  
DbFileSysInfo (&FileSysInfo);  
NumBefore = FileSysInfo.NumOfClean;  
// do some file system operations  
DbFileSysInfo (&FileSysInfo);  
NumOfCleanups = FileSysInfo.NumOfClean - NumBefore;
```

DbFileWrite

Writes bytes to a file.

```
STATUS DbFileWrite(
    FileType file,
    const void * data,
    unsigned size)
```

Parameters

<code>file</code>	A file number.
<code>data</code>	Pointer to the data to be written to the file. This cannot be a direct data pointer to flash memory.
<code>size</code>	Size of the data, in bytes.

Returns One of the following return codes:

```
DB_OK
DB_ERR_BAD_FILE
DB_ERR_FILE_CLOSED
DB_ERR_NO_HANDLE
DB_ERR_NO_SPACE
```

Description The database updates performed by this function are not atomic, and they are not guaranteed to be permanent until the subsequent `DbFileClose` returns `DB_OK`. If a system error occurs, the database can be left in a partially updated state. The update order is not specified. You must ensure that the data is correctly interpreted.

The file extension is subject to the amount of free space available for all the new records required, each of a size `newRecSize`. The system does not attempt to create a smaller record if there is not enough space; instead, `DB_ERR_NO_SPACE` is returned.

This function writes the supplied data to the file, starting at the current position. It overwrites data in previously created database records. The previous record structure is preserved and the record lengths are not changed, unless the database needs to be extended. In this case, the appended records have the size specified by the `newRecSize` parameter of the `DbFileOpen` function. The last appended record might be only partially used, with the consequence of `Length` being smaller than the database length. This is important because the `Length` is not stored permanently, unless `DbFileClose` is performed with `trim` equal to `true`.

DbFindNext

Retrieves a handle to the next database.

```
HandleType DbFindNext(  
    HandleType db,  
    const char * pattern)
```

Parameters

db	A handle to a database indicating the point after which the database directory is to be searched. A negative value indicates that the search is to begin with the first directory entry.
pattern	A pointer to a pattern used to filter the names of the databases in the directory. The asterisk (*) wildcard character matches any number of characters, including no characters. A literal asterisk character is represented as * and the backslash character is represented as \\.

Returns A non-negative handle to the next database matching the pattern, or one of the following negative error codes:

```
DB_NO_DB  
DB_ERR_BAD_HANDLE  
DB_ERR_NOT_DIR
```



Warning: The handle returned by this function persists only for the lifetime of the database on the particular system; it is intended for temporary database identification only. Do not store record or database handles in the permanent data because the data cannot be meaningfully copied to another system.

Description Directory entries are ordered according to the time of creation. This order persists across handheld resets.

The following sample code demonstrates the use of this function:

```
// Try to construct a unique database name.  
char name = "temp0.dat";  
while (DbFindNext (-1, name) != DB_NO_DB) {  
    if (name[4] = '9')  
        return ERROR_COND;  
    name[4]++;  
}  
HandleType newDb = DbGetHandle (name);
```

DbFirstRec

Retrieves a handle to the first database record.

```
HandleType DbFirstRec(HandleType db)
```

Parameters db A database handle.

Returns A non-negative handle to the first record in the database, or one of the following negative error codes:

```
DB_NO_REC
DB_ERR_BAD_HANDLE
DB_ERR_NOT_DIR
```



Warning: The handle returned by this function persists only for the lifetime of the database on the particular system; it is intended for temporary database identification only. Do not store record or database handles in the permanent data because the data cannot be meaningfully copied to another system.

Description This function can also be used on the database of directory entries. Doing so returns the handle to the first database entry in the directory.

DbFreeRec

Retrieves number of free handles.

```
unsigned DbFreeRec()
```

Returns The number of unused handles.

Description Each new database and each new record uses exactly one handle. It is possible to use all of the available handles.

The following sample code demonstrates the use of this function:

```
while (NewData (&Data)) {
    if (DbFreeRec () == 0)
        DeleteOldData ();
    DbAddRec (MyDB, DataSize, &Data);
}
```

DbFreeSpace

Retrieves the number of unused bytes in a database.

```
unsigned DbFreeSpace()
```

Returns The total number of unused flash memory bytes in the file system.

Description You cannot create a record as large as the total free space due to system overhead and flash memory fragmentation. DbFreeSpace can indicate space management problems.

The following sample code demonstrates the use of this function:

```
if (DbFreeSpace () < MinFreeSpace)
```

Chapter 3: File System API

```
EmptyUserTrash ();
```

DbGetHandle

Retrieves a handle to the database, or creates a handle if one does not exist.

```
HandleType DbGetHandle(const char * fileName)
```

Parameters `fileName` The name of the database to be created or accessed.

Returns A non-negative handle to the named database, or one of the following negative error codes:

```
DB_ERR_NO_SPACE  
DB_ERR_NO_HANDLE  
DB_ERR_BAD_NAME
```



Warning: The handle returned by this function persists only for the lifetime of the database on the particular system; it is intended for temporary database identification only. Do not store record or database handles in the permanent data because the data cannot be meaningfully copied to another system.

Description This is the only function that can create a new database (or a streamed file). If the named database does not exist then it is created and assigned a new handle.

The cost of determining the database handle from a string representation of the database name is 10 to 100 times the cost of referencing a single record given the database handle. As a result, application programs should typically retain the handles to databases when they are being actively used.

The theoretical maximum length of a database name is $2^{16}-5$ characters, or the maximum record size and a small amount of overhead. The database name is stored as a record in the database.

The following sample code demonstrates the use of this function:

```
HandleType myDb, newRec; myDb = DbGetHandle ("timestamps");  
  
for (i = 0; i < 150; i++)  
newRec = DbAddRec (myDb, sizeof (long), RimGetTicks ());
```

DbMaxHandles

Retrieves the maximum number of record and file handles.

```
void DbMaxHandles(
    int * MaxRecHandles,
    int * MaxFileHandles)
```

Parameters	MaxRecHandles	A pointer to a 32-bit integer variable where the maximum number of record handles will be stored. A record handle is used to identify individual records, including database header records, within the file system.
	MaxFileHandles	A pointer to a 32-bit integer variable where the maximum number of file handles will be stored. A file handle is used to identify a database currently open for streamed access.

DbMaxNewRecSize

Retrieves the maximum possible size of a new record.

```
unsigned DbMaxNewRecSize(RecKind NewRec)
```

Parameters	NewRec	RecKind is an enumerated type, so NewRec is the kind of record to be created; one of: NORMAL — a normal database record ORPHAN — an orphan record DB_NAME — a database directory entry
-------------------	--------	---

Returns The maximum possible size of a new record of the specified kind, in bytes.

The following sample code demonstrates the use of this function:

```
if (DbMaxNewRecSize (NORMAL) < DataSize)
    MustSliceData ();
```

DbName

Retrieves the database name from a handle.

```
const char * DbName(HandleType db)
```

Parameters db A handle to a database.

Returns A pointer to a read-only, NULL-terminated character string that contains the name of the database, or NULL if db is not a valid database handle.

The following sample code demonstrates the use of this function:

```
// Collect all database names starting with "temp"
HandleType currDb;
currDb = DbFindNext (-1, "temp*");

while (currDb >= 0) {
    char * name = DbName (currDb);
    collect (name);
    currDb = DbFindNext (currDb, "temp*");
}
```

DbNextRec

Retrieves the handle to the next database record.

```
HandleType DbNextRec(HandleType prevRec)
```

Parameters prevRec A handle to a database record.

Returns A non-negative handle to the next record in the database, or one of the following negative error codes:

```
DB_NO_REC
DB_ERR_BAD_HANDLE
```



Warning: The handle returned by this function is intended for temporary database identification only; it persists for the lifetime of the database on the particular system. Do not store record or database handles in the permanent data because the data cannot be meaningfully copied to another system.

Description This function returns the handle to the record immediately following the record specified by prevRec. An orphan record does not have a successor.

The DbNextRec function can also be applied to a database handle. In this case, the result of the function is the handle to the next database directory entry.

DbPointTable

Retrieves a pointer to the handle mapping table.

```
const void * const * DbPointTable()
```

Returns Pointer to the handle mapping table.



Warning: The contents of this table are not guaranteed to persist unchanged across any operation which might result in a change to the file system's permanent data or its organization, such as cleanup of dirty sectors, file system calls, and yielding the thread to other applications. Use any stored values from this table carefully.

Description Each record handle can be used as an index into this table to obtain a direct pointer to the contents of the record.

For a database handle, the associated pointer references the directory entry. Using directory entry data directly is not advised since its format can change in future versions of the software.

The location of this table does not change until the file system software is reloaded.

DbPointTableEdition

Retrieves the edition counter value.

```
DWORD DbPointTableEdition()
```

Returns The current value of the edition counter.

Description The `DbPointTableEdition` function is for use with the `PointerTable`; refer to the *Developer Guide* for details on how to use this function.

DbRecSize

Retrieves the size of a database record.

```
int DbRecSize(
    HandleType rec,
    BOOL dataOnly)
```

Parameters

<code>rec</code>	The handle to the record for which the size is requested.
<code>dataOnly</code>	If true, report only the size of the user data in the record. If false, the file system overhead is also included.

Returns A non-negative record size, in bytes, or the following negative error code:

```
B_ERR_BAD_HANDLE
```

Description This function is fast enough that generally you do not need to keep track of the length of each record.

DbReplaceOrphan

Replaces a database record with an orphan record.

```
STATUS DbReplaceOrphan(  
    HandleType rec,  
    HandleType orphan)
```

Parameters

rec	The handle to the record to be replaced.
orphan	The handle to the orphan record to replace rec.

Returns One of the following return codes is returned:

```
DB_OK  
DB_ERR_NO_SPACE  
DB_ERR_BAD_HANDLE  
DB_ERR_IS_DIR  
DB_ERR_NOT_ORPHAN
```

Description This function is atomic. The record is either replaced or the database is unchanged. No intermediate state is visible. Data consistency is preserved if a system error occurs when the function is executing.

This function replaces a record in a database with an existing orphan record. It is very efficient because it does not extend the log and does not cause a cleanup.

You cannot use this function to replace database directory entries. For security reasons, these entries should only be modified by the file system.

DbReplaceRec

Replaces database record with a new one.

```
STATUS DbReplaceRec(  
    HandleType rec,  
    unsigned newSize,  
    const void * data)
```

Parameters

rec	The handle to the record to be replaced.
newSize	The size of the new data, in bytes. The value must be between 0 and $2^{16}-2$, inclusive. At any particular instant, the maximum possible record size is additionally limited due to flash memory fragmentation. Refer to "DbMaxNewRecSize" on page 61 for more information.
data	A pointer to the data to be written into the record. If it is NULL, the data is assumed to have all bits set to one. This can be used with the DbAndRec function.

Returns One of the following return codes:

```
DB_OK
DB_ERR_BAD_SIZE
DB_ERR_NO_SPACE
DB_ERR_BAD_HANDLE
DB_ERR_IS_DIR
```

Description This function is atomic. The record is either completely replaced or it remains unchanged. No intermediate state is visible. Data consistency is preserved if a system error occurs when the function is executing.

You can also call `DbAndRec` instead of `DbReplaceRec`. `DbAndRec` is faster, but it is not atomic.

You cannot use this function on database directory entries. For security reasons, these entries should only be modified by the file system.

The data parameter can be a pointer to some other portion of the file system. However, the result is about 7% slower than if the data resides in RAM because flash to flash copying must be buffered through RAM.

DbSecure

Overwrites deleted data with zeroes.

```
void DbSecure()
```

Description `DbSecure` overwrites all dirty records with zeroes to ensure that sensitive data is not left in the file system.

DbSize

Retrieves database size.

```
int DbSize(
    HandleType db,
    BOOL dataOnly)
```

Parameters

<code>db</code>	A database handle.
<code>dataOnly</code>	If true, report only the size of the user data in the database. If false the file system overhead is included.

Returns A non-negative database size, in bytes, or one of the following negative error codes:

```
DB_ERR_BAD_HANDLE
DB_ERR_NOT_DIR
```

The following sample code demonstrates the use of this function:

```
// Create indices for sorting databases by size
```

Chapter 3: File System API

```
HandleType handles[MAX_NUM_DB];
int sizes[MAX_NUM_DB];
int i = 0;
HandleType CurrDb = DbFindNext (-1, NULL);
while (CurrDb >= 0) {
    int Temp = DbSize (CurrDb, TRUE);
    assert (Temp >= 0);
    handles[i] = CurrDb;
    sizes[i] = Temp;
    i++;
    CurrDb = DbFindNext (currDb, NULL);
}
```

Return codes

Database/file system functions return the following return codes, shown here in alphabetical order:

Return code	Description
DB_ERR_BAD_FILE	The file parameter is an invalid file number.
DB_ERR_BAD_HANDLE	The db, rec, prevRec or orphan parameter is not a valid handle.
DB_ERR_BAD_NAME	An invalid database name was entered.
DB_ERR_BAD_SIZE	The specified record size is invalid; it must be between 1 and $2^{16}-2$, inclusive.
DB_ERR_BOUNDS	The region to be modified does not fit within the existing record.
DB_ERR_FILE_CLOSED	The specified file is not open.
DB_ERR_FILE_OPEN	The streamed file access is currently open on the specified database. It must be closed before deleting the database.
DB_ERR_IS_DIR	The rec parameter refers to a directory entry.
DB_ERR_NO_FILE	No handle is available.
DB_ERR_NO_HANDLE	No handle is available.
DB_ERR_NO_SPACE	There is not enough flash memory to perform the requested action.
DB_ERR_NOT_DIR	The db parameter is not a valid database directory entry.
DB_ERR_NOT_EXIST	The named database does not exist.
DB_ERR_NOT_ORPHAN	The specified record is not an orphan record.
DB_ERR_PRIV_HANDLE	The specified handle is privileged.
DB_ERR_WRONG_DB	The database did not match the record.
DB_NO_DB	No databases with matching names exist after the specified database.
DB_NO_REC	The database has no records. (This is not an error.)
DB_OK	The action was completed successfully.

Chapter 3: File System API

Chapter 4 **Serial communications API**

The Serial Communications API, declared in `comm.h`, provides access to the serial port.

Serial communications functions

The functions on the following pages are listed in alphabetical order.

<code>CommClosePort</code>	70
<code>CommGetDtr</code>	70
<code>CommOpenPort</code>	70
<code>CommReadBuffer</code>	71
<code>CommReadChar</code>	71
<code>CommRegisterNotifyPattern</code>	72
<code>CommSendBuffer</code>	73
<code>CommSendChar</code>	73
<code>CommSetDsr</code>	73
<code>CommSetFlowControl</code>	74
<code>CommSettings</code>	74
<code>CommStandbyMode</code>	74
<code>CommTxCount</code>	75

CommClosePort

Closes the COM port, disables line drivers, and frees any buffer memory that was allocated when the COM port was opened.

```
void CommClosePort()
```

Description This function closes the COM port, disables the line drivers, and frees any buffer memory that was allocated when the COM port was opened. When the handheld is shut off, the system calls `CommClosePort` automatically. Applications must not assume that the COM port is still open once the handheld is powered on again.

For example, see `comtest.c`.

CommGetDtr

Reads state of host DTR modem control line.

```
int CommGetDtr()
```

Returns The current state of the DTR control line.

Description The DTR line is actually the DTR control line as driven by an attached PC.

CommOpenPort

Allocates serial and transmit first-in, first-out (FIFO) buffers in memory, enables the serial driver and hardware serial line drivers.

```
int CommOpenPort(  
    DWORD baud,  
    int databits,  
    int parity,  
    int Stopbits,  
    int RxBufferSize,  
    int TxBufferSize)
```

Parameters	baud	The baud rate that the serial port is to use.
	databits	The number of bits to use (7 or 8).
	parity	The parity method to use (one of: <code>COMM_NO_PARITY</code> , <code>COMM_EVEN_PARITY</code> , or <code>COMM_ODD_PARITY</code>).
	Stopbits	The number of stop bits to use (1 or 2).
	RxBufferSize	The receive FIFO serial buffer size. It must be a power of two. Values that are not a power of two are rounded up to a power of two.
	TxBufferSize	The transmit FIFO serial buffer size. Values are rounded up to a power of two.

Returns True if successful; false otherwise.

Description This function allocates serial and transmit FIFO buffers in memory, and enables the serial driver, as well as hardware serial line drivers. It is advisable to always close the COM port when it is not in use, as the hardware line drivers consume extra power while the port is open.

Example See `comtest.c`.

CommReadBuffer

Read data received over the serial port.

```
int CommReadBuffer(
    BYTE * Data,
    int Length)
```

Parameters

Data	A pointer to a buffer for holding the received data. Set this to NULL if the data is to be discarded. This can be used to flush the receive buffer.
-------------	---

Length	The maximum number of bytes to remove from the receive buffer.
---------------	--

Returns The number of bytes actually read, or 0 (zero) if data is NULL.

Description The number of bytes read is the number of bytes in the serial receive FIFO when the function is called. `CommReadBuffer` only returns the bytes received when `CommReadBuffer` was called, and does not wait for extra bytes to arrive. If `CommReadBuffer` returns less than what was passed in as `Length`, the serial receive buffer is guaranteed to be empty after the call. Emptying the buffer ensures that a new `COMM_RX_AVAILABLE` message is generated once new bytes arrive.

If data is NULL, `CommReadBuffer` flushes the receive buffer to empty.

CommReadChar

Read one byte from the serial port.

```
int CommReadChar()
```

Returns The byte read, or negative if an error occurred or no bytes were available.

Description This function is identical to calling `CommReadBuffer` with a length of one.

For example, see `comtest.c`.

CommRegisterNotifyPattern

Allows multiple applications to register a pattern to look for on the serial port.

```
DWORD __cdecl CommRegisterNotifyPattern(void * NotifyPattern)
```

Parameters `NotifyPattern` This parameter points to a 4-byte string containing a pattern. The pattern's first byte must be non-zero, and is always 4 bytes in length.

Returns One of the following return codes:

0	COMM_NOTIFY_REGISTERED_NEW
1	COMM_NOTIFY_ALREADY_REGISTERED_BY_SELF
2	COMM_NOTIFY_ALREADY_REGISTERED_BY_OTHER
3	COMM_NOTIFY_REPLACED_OLDEST
4	COMM_NOTIFY_DEREGISTERED_SINGLE
5	COMM_NOTIFY_DEREGISTERED_ALL
6	COMM_NOTIFY_NOT_REGISTERED
7	COMM_NOTIFY_NOT_REGISTRATION_OWNER

Description This function allows multiple applications to register a pattern to look for on the serial port. While DTR is active, the port is monitored at 9600 baud 8N1. If the specified pattern occurs, the application that registered that pattern is sent a COMM_PATTERN_NOTIFY event.

Applications can de-register patterns by calling `CommRegisterNotifyPattern` with a NULL parameter. Applications can only register one pattern at a time. Registering a second pattern cancels the first.

The following examples demonstrates the use of this function:

- `CommRegisterNotifyPattern("abcd")`
This causes a `CommRegisterNotifyPattern` message to be sent to the calling task when the pattern `abcd` is seen on the port.
- `CommRegisterNotifyPattern("a")`
This call is not legal, as the `Pattern` parameter points only to an "a" followed by zero followed by two unknown characters.

See also `comtest.c`.

CommSendBuffer

Places the bytes to be sent into the transmit serial FIFO.

```
int CommSendBuffer(
    const void * Data,
    int Length)
```

Parameters

Data	A pointer to the data to be sent.
Length	The number of bytes to be sent.

Returns The number of bytes that could be placed in the transmit FIFO, or negative if an error occurred. `CommSendBuffer` places the data in the transmit FIFO and start transmitting, but returns before the data has left the port.

Description This function places the bytes to be sent into the transmit serial FIFO. The function returns immediately, and indicates how many bytes were actually placed in the buffer. If an item larger than the serial buffer is to be sent, it must be split into several chunks, while waiting for the `COMM_TRANSMIT_EMPTY` message in between. For example, see `comtest.c`.

CommSendChar

Returns the number of bytes that could be placed in the transmit FIFO.

```
int CommSendChar(BYTE Character)
```

Parameters

Character	The single character to be sent.
-----------	----------------------------------

Returns The number of bytes that could be placed in the transmit FIFO. It returns negative if an error occurred.

Description This function is identical to calling `CommSendBuffer` with only one byte.

CommSetDsr

Set DSR modem control line.

```
void CommSetDsr(int State)
```

Parameters

State	The desire state for the DSR signal (0 or 1).
-------	---

Description The DSR signal is presented as DSR to an attached PC.

CommSetFlowControl

Sets the type of flow control to use for serial communications.

```
void CommSetFlowControl(enum FlowControlType flow)
```

Parameters flow One of the following values:
SERIAL_NO_FLOW_CONTROL
SERIAL_SOFTWARE_FLOW_CONTROL
SERIAL_RTS_CTS_FLOW_CONTROL

CommSettings

Change the configuration of the serial port.

```
void CommSettings(  
    DWORD Baud,  
    int Databits,  
    int Parity,  
    int Stopbits)
```

Parameters

Baud	The baud rate that the serial port is to use.
Databits	The number of bits to use (7 or 8).
Parity	The parity method to use (one of COMM_NO_PARITY, COMM_EVEN_PARITY, COMM_ODD_PARITY)
Stopbits	The number of stop bits to use (1 or 2).

Description This function is used to change the configuration of the serial port. For example, see `comtest.c`.

CommStandbyMode

Puts the COM port into standby mode.

```
void CommStandbyMode(BOOL State)
```

Parameters State Calling with true puts the COM port into standby mode; calling with false disables standby mode.

Description This function keeps the COM port open, but allows the serial drivers and processor to go into suspend mode. The caller must already have the COM port open, and must remember to call `CommStandbyMode (FALSE)` before using the COM port again.

CommTxCount

Returns the number of characters remaining to be transmitted in the serial transmit FIFO.

```
int CommTxCount()
```

Returns The number of characters in the transmit buffer.

Description The number of characters remaining to be transmitted in the serial transmit FIFO; 0 (zero) if the transmit FIFO and hardware are completely empty.

Constants

The following constants are specific to the serial communications API.

These constants are used by `CommOpenPort` and `CommSettings`.

Constant	Description
COMM_NO_PARITY	Bytes are to be framed without a parity bit.
COMM_EVEN_PARITY	Bytes are to be framed with even bit parity.
COMM_ODD_PARITY	Bytes are to be frames with odd bit parity.

Error Codes

Error code	Description
COMM_ERROR_NOT_OPEN	This value is returned on COM function calls when the port is not open.
COMM_ERROR_PARITY	Indicates that a receive error has occurred.
COMM_ERROR_FRAMING	Indicates that a frame error has occurred.
COMM_ERROR_OVERRUN	Indicates the COM buffer had a buffer overrun.
COMM_ERROR_BREAK	Indicates a COM break was received.

Events

The following device events are described in “COM device events” on page 119:

COMM_CONTROL_CHANGE
COMM_PATTERN_NOTIFY
COMM_RX_ERROR
COMM_RX_AVAILABLE
COMM_TX_EMPTY

Chapter 5

Keypad API

The Keypad API functions, declared in `Keypad.h`, provide control of the handheld keyboard.

Functions

The functions on the following pages are listed in alphabetical order.

Keypad::KeypadBeep	77
Keypad::KeypadRate	77
Keypad::KeypadRegister	78

Keypad::KeypadBeep

Turns key tones on or off when a key is pressed.

```
void KeypadBeep(BOOL Enable)
```

Parameters	Enable	If <code>Enable</code> is true, a tone is generated when a key is pressed; otherwise, the key tone is disabled.
-------------------	--------	---

Keypad::KeypadRate

Configures the keypad for auto key repeat.

```
void KeypadRate( WORD Delay, WORD Rate)
```

Parameters	Delay	The delay, in milliseconds, after a key is held down before it begins to repeat.
	Rate	The rate, in milliseconds, at which <code>KEY_REPEAT</code> messages are sent to an application; a value of 0 disables the auto repeat key feature.

Chapter 5: Keypad API

Description This function configures the keypad for auto key repeat. The `Delay`, in milliseconds, is the initial delay before the first `KEY_REPEAT` event is sent while the key is held down.

The `Rate`, also in milliseconds, is the rate at which `KEY_REPEAT` events are sent to an application after the initial delay. The auto repeat feature is disabled if `Rate` is 0. By default, the auto repeat key feature is enabled.

Keypad::KeypadRegister

Enables an application to register to receive events when hot keys are pressed.

```
BOOL KeypadRegister(DWORD Key)
```

Parameters `Key` The keystroke which the application desires to intercept on a global basis. A valid value consists of a key identification plus an optional modifier. The key identification can be any capital letter, `KEY_BACKSPACE`, `KEY_SPACE`, `KEY_ENTER`, or `THUMB_CLICK`. The modifier can be either `(SHIFT_STATUS << 16)` or `(ALT_STATUS << 16)`.

Returns True if successful; false if there was insufficient space in the key registry to register this request.

Description This function allows a system services application to intercept global hot keys. These hot keys are sent only to the registered handler, and not to the foreground application. For example, an application that provides an improved Task Switcher and power-off facility could intercept the `SHIFT + CLICK` or `ALT + ENTER` combinations with the following code:

```
KeypadRegister( (SHIFT_STATUS << 16) | THUMB_CLICK );  
KeypadRegister( (ALT_STATUS << 16) | KEY_ENTER );
```

When the user invokes either of these two events, the keypad message is sent to this application, regardless of which application was in the foreground. For the application that intercepts the hot key to interact with the user, it must call `RequestForeground` to bring itself to the foreground.

Constants

Constant	Meaning
ALT_LOCK	For all keypad events, if (Data[0] and ALT_LOCK) is true, then the ALT LOCK key was on during the event.
ALT_STATUS	For all keypad events, if (Data[0] and ALT_STATUS) is true, then the ALT key was pressed during the event.
CAPS_LOCK	For all keypad events, if (Data[0] and CAPS_LOCK) is true, then the CAPS LOCK key was on during the event.
KEY_HELD_WHILE_ROLLING	For THUMB_ROLL_UP/DOWN events, if (Data[0] and KEY_HELD_WHILE_ROLLING) is true, then the last key combination pressed is being held while the thumb wheel is being rolled.
SHIFT_STATUS	For all keypad events, if (Data[0] and SHIFT_STATUS) is true, then the SHIFT key was pressed during the event.

The following constants define some ASCII and some device-specific key codes.

Constant	Meaning
KEY_ALT	This defines the scan code of the handheld ALT key.
KEY_BACKSPACE	Standard ASCII backspace: 0x08.
KEY_DELETE	Standard ASCII delete: 0x7F.
KEY_ENTER	Standard ASCII enter: 0x0D
KEY_SHIFT	This defines the scan code of the handheld SHIFT key.
KEY_SPACE	Standard ASCII space: 0x20.

Events

The following device events are described in “KEYPAD device events” on page 115:

KEY_DOWN
 KEY_REPEAT
 KEY_UP
 THUMB_CLICK
 THUMB_UNCLICK
 THUMB_ROLL_UP
 THUMB_ROLL_DOWN
 KEY_STATUS

Chapter 5: Keypad API

Chapter 6

LCD API

LCD functions, declared in `LCD_API.h`, provide control of the handheld LCD display.

Structures

LcdConfig

```
typedef struct {  
    WORD    LcdType;  
    WORD    contrastRange;  
    WORD    width;  
    WORD    height;  
} LcdConfig;
```

Field	Description
LcdType	Reserved.
contrastRange	Maximum contrast value for <code>LcdGetContrast</code> and <code>LcdSetContrast</code> .
width	Width of the LCD in pixels.
height	Height of the LCD in pixels.

LCD functions

The functions in the following pages are listed in alphabetical order.

LcdClearDisplay	82
LcdClearToEndOfLine	83
LcdCopyBitmapToDisplay	83
LcdCopyDisplay	84
LcdCreateDisplayContext	84
LcdDestroyDisplayContext	85
LcdDrawBox	85
LcdDrawLine	86
LcdForceRefresh	87
LcdGetCharacterWidth	87
LcdGetConfig	88
LcdGetContrast	88
LcdGetCurrentFont	88
LcdGetCursorPoint	88
LcdGetDisplayContext	89
LcdGetFontHeight	89
LcdGetFontName	91
LcdGetNumberOfFonts	91
LcdGetPixel	92
LcdGetRegion	92
LcdGetStringWidth	93
LcdIconsEnable	93
LcdPutStringXY	94
LcdRasterOp	96
LcdReplaceFont	99
LcdScroll	99
LcdSetContrast	100
LcdSetCurrentFont	100
LcdSetCursorPoint	101
LcdSetDisplayContext	101
LcdSetPixel	102
LcdSetRegion	102
LcdSetTextWindow	103

LcdClearDisplay

Clears the display by turning all pixels off.

```
void LcdClearDisplay()
```

Description This function clears the display by turning all pixels off, then places the cursor at the top left-hand corner of the display. If a text window is defined, only the text window is cleared.

LcdClearToEndOfLine

Turns off all pixels to the right of the current cursor position.

```
void LcdClearToEndOfLine()
```

Description This function turns off all pixels to the right of the current cursor position, beginning at the current row of pixels and extending downward the current font height. (The current font height can be determined with the `LcdGetCurrentFontHeight` function, but this is not necessary when using `LcdClearToEndOfLine`.)

For example, see `comtest.c` and `regress.c`.

The following sample code demonstrates how to use this function:

```
// Display a line of text, move the cursor back, then turn off all pixels
// to the end of the line
LcdCenterString("MESSAGE SENT");
for (x=1;x<=5;x++)
    moveCursor(LEFT);
LcdClearToEndOfLine();
```



Result of `LcdClearToEndOfLine` sample code

LcdCopyBitmapToDisplay

Copies the entire bitmap defined in the structure pointed to by `pbmSource` to the display.

```
void LcdCopyBitmapToDisplay(
    const BitMap * pbmSource,
    int iDisplayX,
    int iDisplayY)
```

Parameters	<code>pbmSource</code>	A pointer to a bitmap which is to be copied to the current virtual display.
	<code>iDisplayX</code>	The horizontal location of the upper left corner to which the bitmap will be copied.
	<code>iDisplayY</code>	The vertical location of the upper left corner to which the bitmap will be copied.

Description This function copies the entire bitmap defined in the structure pointed to by `pbmSource` to the display. An offset in the display can be specified using `iDisplayX` and `iDisplayY`.

The following sample code demonstrates how to use this function:

```
// Displays a clock bitmap to the screen at coordinate x =5, y =15.
```

```
LcdCopyBitmapToDisplay( &Clock, 5, 15 );
```

See also `clock.c`.

LcdCopyDisplay

Copies the display buffer of an existing display context to another existing display.

```
int LcdCopyDisplay(  
    int iSourceDC,  
    int iDestDC)
```

Parameters

<code>iSourceDC</code>	The number of the source display context.
<code>iDestDC</code>	The number of the destination display context.

Returns LCD_OK if successful and LCD_BAD_HANDLE if display context is invalid.

Description This function copies only the display buffer. No font attributes are copied.

The following sample code demonstrates how to use this function:

```
// Copy the display buffer of display #2 to display #3.  
LcdCopyDisplay ( 2, 3 );
```

LcdCreateDisplayContext

Creates a new display context, then copies the contents of an existing display context into it.

```
int LcdCreateDisplayContext(int iDisplayToCopy)
```

Parameters

<code>iDisplayToCopy</code>	The number of the existing display context that is to be copied into the new display context. Using a value of -1, the display context uses the system defaults.
-----------------------------	--

Returns The number assigned to the new display context.

Description This function creates a new display context, then copies the contents of an existing display context into it. A display context is a buffer which holds display information; only one display can be active at a time. This is useful for preserving the display when task-swapping.

The following sample code demonstrates how to use this function:

```
// Copy the contents of the current display context into a new display  
// context, then clear the current display context.  
// Write some text on the screen, then bring up the previous display.
```

```
LcdCenterString( "Original application", 10 );  
current = LcdGetDisplayContext();  
new = LcdCreateDisplayContext( current );
```

```
// Note that display context #current is still active
```

```
LcdClearDisplay();
LcdCenterString( "We replaced the text", 10 );

// Now we switch to the copy of the original display, which remained intact
LcdSetDisplayContext( new );
```

LcdDestroyDisplayContext

Removes a display context from memory.

```
int LcdDestroyDisplayContext(int iDC)
```

Parameters iDC The number of the display context.

Returns LCD_OK if the display context was successfully destroyed; LCD_BAD_HANDLE if an invalid display context is chosen.

Description A display context is a buffer that holds display information; only one display can be active at a time. This is useful for preserving the display when task-swapping.

The following sample code demonstrates how to use this function:

```
// Remove display context #3 from memory.
LcdDestroyDisplayContext( 3 );
```

LcdDrawBox

Draws a hollow box on the display.

```
void LcdDrawBox(
    int iDrawingMode,
    int x1,
    int y1,
    int x2,
    int y2)
```

Parameters iDrawingMode Acceptable values are DRAW_WHITE, DRAW_INVERT, DRAW_BLACK.

x1 The horizontal pixel of the top left corner of the box.

y1 The vertical pixel of the top left corner of the box.

x2 The horizontal pixel of the bottom right corner of the box.

y2 The vertical pixel of the bottom right corner of the box.

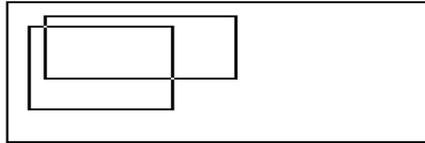
Description This function draws a hollow box on the display, two diagonally opposite corners of which are (x1, y1) and (x2, y2). If iDrawingMode = DRAW_WHITE, the pixels are turned off. If iDrawingMode = DRAW_BLACK, the pixels are turned on. If iDrawingMode = DRAW_INVERT, off pixels are turned on and on pixels are turned off.

Chapter 6: LCD API

See `clock.c`.

The following sample code demonstrates how to use this function:

```
// Draw two boxes
LcdDrawBox( DRAW_BLACK, 5, 10, 50, 50 );
LcdDrawBox( DRAW_INVERT, 10, 5, 70, 35 );
```



Result of `LcdDrawBox` sample code

LcdDrawLine

Draws a line on the display.

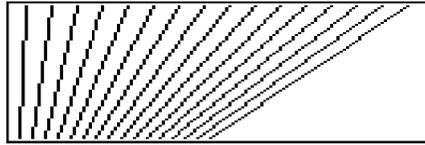
```
void LcdDrawLine(
    int iDrawingMode,
    int x1,
    int y1,
    int x2,
    int y2)
```

Parameters	<code>iDrawingMode</code>	Acceptable values are <code>DRAW_WHITE</code> , <code>DRAW_INVERT</code> , <code>DRAW_BLACK</code> .
	<code>x1</code>	The horizontal pixel of the line's starting point.
	<code>y1</code>	The vertical pixel of the line's starting point.
	<code>x2</code>	The horizontal pixel of the line's ending point.
	<code>y2</code>	The vertical pixel of the line's ending point.

Description This function draws a line on the display, the two ends of which are (`x1`, `y1`) and (`x2`, `y2`). If `iDrawingMode = DRAW_WHITE`, the pixels are turned off. If `iDrawingMode = DRAW_BLACK`, the pixels are turned on. If `iDrawingMode = DRAW_INVERT`, off pixels are turned on and on pixels are turned off.

The following sample code demonstrates how to use this function:

```
// Draw a series of lines
for(x=2; x<=65; x+=4)
    LcdDrawLine(DRAW_BLACK, x, 64, x*2, 0);
```



Result of LcdDrawLine sample code

See also `bouncer.c`.

LcdForceRefresh

Copies the application's virtual buffer to the display.

```
void LcdForceRefresh()
```

Description This function copies the application's virtual buffer to the display. In order to speed up the display rendering, all of an application's LCD functions render into the application's private virtual display buffer. No screen updates are visible until the application calls `LcdForceRefresh` or yields control (for example, by calling `RimGetMessage`).

The following sample code demonstrates how to use this function:

```
LcdPuts("BOOM!" );
LcdForceRefresh();
// this copies the virtual buffer to the display
DoLengthyOperation();
LcdClearDisplay();
```

During the lengthy operation, the screen shows this:



Result of LcdForceRefresh sample code

LcdGetCharacterWidth

Gets width of LCD display character for a particular font, in pixels.

```
int LcdGetCharacterWidth(
    char c,
    int iFontIndex)
```

Parameters	<code>c</code>	The character for which width is required.
	<code>iFontIndex</code>	The index number of the font in which the character would be drawn. Valid numbers are from 0 to 4. As a special case, specifying the <code>fontIndex</code> as <code>-1</code> uses the currently selected font.

Chapter 6: LCD API

Returns Width (in pixels) of the specified character, if it was drawn in the specified font, or LCD_BAD_FONT_INDEX if the font index specified was invalid.

Description The return value is the horizontal width (number of pixels) of the specified character drawn in the specified font.

The following sample code demonstrates how to use this function:

```
// Get width (in pixels) of characters in current font.
int size
size = LcdGetCurrentFontWidth();
```

LcdGetConfig

Determines properties of the handheld LCD.

```
void LcdGetConfig(LcdConfig * config)
```

Parameters config A pointer to an LcdConfig structure that receives the LCD configuration information.

Description Use this function to determine the height, width, and contrast of the LCD. For example, see HELLO.C.

LcdGetContrast

Determines the current contrast setting of the LCD.

```
int LcdGetContrast()
```

Returns The current contrast setting of the LCD. This value is a number from 0 (lightest) to the maximum contrast (darkest) setting supported which is LCD_MAX_CONTRAST.

Description This function determines the current contrast setting of the LCD. The contrast can be changed using the LcdSetContrast function.

LcdGetCurrentFont

Returns the font index that was last selected by LcdSetCurrentFont for the current display context.

```
int LcdGetCurrentFont()
```

Returns The index number of fonts that are used for future text display. Valid index numbers range from 0 to 4.

For example, see bouncer.c.

LcdGetCursorPoint

Determines the current pixel position of the cursor.

```
void LcdGetCursorPoint(
    int *xPixel,
    int *yPixel)
```

Parameters

`xPixel` The current horizontal pixel position of the cursor (0=left).

`yPixel` The current vertical pixel position of the cursor (0=top).

Description This function determines the current pixel position of the cursor, and places the horizontal position (from left) in `xPixel` and the vertical position (from top) in `yPixel`.

The following sample code demonstrates how to use this function:

```
// Move cursor 10 pixels to the left and 20 pixels down.
int cursorX, cursorY;
LcdGetCursorPoint(&cursorX, &cursorY);
LcdSetCursorPoint(cursorX - 10, cursorY + 20);
```

LcdGetDisplayContext

Retrieves the handle of the display context that is currently activated by the task.

```
int LcdGetDisplayContext()
```

Returns The handle of the current display context. If there is an error, the function returns `LCD_BAD_HANDLE`.

Description A display context is a buffer which holds display information; only one display can be active at a time. This is useful for preserving the display when task-swapping.

The following sample code demonstrates how to use this function:

```
// Store the handle of the current display context.
int CurrentDC;
CurrentDC = LcdGetDisplayContext();
```

See also `bouncer.c` and `clock.c`.

LcdGetFontHeight

Retrieves the vertical height of each character of specified font.

```
int LcdGetFontHeight(int iFontIndex)
```

Parameters

`iFontIndex` The index number of the font for which height is required. Valid index numbers range from 0 to 4. As a special case, specifying the `iFontIndex` as `-1` gets the height of the currently selected font.

Returns The height (in pixels) of the characters in the specified font or `LCD_BAD_FONT_INDEX` if the font index specified was invalid.

Example `// Get height (in pixels) of characters in current font.`

Chapter 6: LCD API

```
int size;  
size = LcdGetFontHeight( FONT_BOLD );
```

See also `hello.c`, `realtime.c`, and `regmess.c`.

LcdGetFontName

Retrieves the name of the specified font.

```
int LcdGetFontName(
    int iFontIndex,
    const char ** fontName)
```

Parameters

<code>iFontIndex</code>	The index number of the font for which height is required. Valid index numbers range from 0 to <code>LcdGetNumberOfFonts()</code> . As a special case, specifying the <code>fontIndex</code> as <code>-1</code> gets the height of the currently selected font.
<code>fontName</code>	A pointer to a string that indicates the font name after the function is executed.

Returns `LCD_OK` if `fontIndex` is within the range of currently available fonts (see `LcdGetNumberOfFonts()`), or `LCD_BAD_FONT_INDEX` if the index is out of range.

Description Fonts 2 through 4 are user-defined. They are initially set to `Font8High`. See `LcdReplaceFont()` for information on how to replace fonts 2 through 4 with user-defined fonts.

The following sample code demonstrates how to use this function:

```
// Get the name of the current font.
char *name;
if (LcdGetFontName( -1, &name ) == LCD_OK) {
    // use name
}
```

LcdGetNumberOfFonts

Retrieves the number of fonts available.

```
int LcdGetNumberOfFonts()
```

Returns The number of fonts available.

Description The valid range of font indices is 0 to `LcdGetNumberOfFonts()`. By default, there are `FONT_SET_MAX` fonts installed (numbered 0 through `FONT_SET_MAX-1`). If resource fonts were installed in a resource DLL, the number returned is `FONT_SET_MAX` plus the number of fonts in the resource DLL.

The following sample code demonstrates how to use this function:

```
int numFonts;
//Obtain the total number of fonts installed
numFonts = LcdGetNumberOfFonts();
```

LcdGetPixel

Determines whether the specified pixel is on (black) or off (white).

```
BOOL LcdGetPixel(  
    int x,  
    int y)
```

Parameters

- x The horizontal pixel position (0=left).
- y The vertical pixel position (0=top).

Returns True if the LCD pixel at the specified (x, y) position is on and false if it is off.

Description This function provides a very low-level interface to the LCD display system.

LcdGetRegion

Copy contents of a region of the LCD into a user buffer.

```
void LcdGetRegion(  
    int left,  
    int top,  
    int width,  
    int height,  
    void * BitArray)
```

Parameters

- left The horizontal pixel location of the top left corner of the rectangular area on the LCD.
- top The vertical pixel location of the top left corner of the rectangular area on the LCD.
- width The pixel width of the rectangular area on the LCD.
- height The pixel height of the rectangular area on the LCD.
- BitArray A pointer to space for a sequence of bits, packed 8 to a byte to be read from the rectangular area on the LCD. The least significant bit (LSB) is first.

Description This function is the counterpart to LcdSetRegion.

LcdGetStringWidth

Determines the number of horizontal pixels a string requires.

```
int LcdGetStringWidth(
    const char * sz,
    int iFontIndex,
    int length)
```

Parameters	<code>sz</code>	A pointer to a string whose width in pixels is required.
	<code>iFontIndex</code>	The index number of the font in which the character would be drawn. Valid numbers range from 0 to 4. As a special case, specifying the <code>iFontIndex</code> as -1 uses the currently selected font.
	<code>length</code>	The number of characters in the string. A length of -1 uses all of the characters up to, but not including, a NULL terminator byte.
Returns		The horizontal width, in pixels, of the string pointed to by <code>sz</code> , or <code>LCD_BAD_FONT_INDEX</code> if the font index specified is invalid.
Description		This function is useful for determining the number of horizontal pixels a string would require. The calculation is made using the width of each character based on the specified font. The string itself is not displayed. For example, see <code>regmess.c</code> .

LcdIconsEnable

Enables or disables the ALT or SHIFT key icon.

```
void LcdIconsEnable(BOOL Enable)
```

Parameters	<code>Enable</code>	Set this parameter to true to enable the icons or false to disable the icons.
Description		This function is used to enable or disable the ALT or SHIFT key icon that appears at the top right corner of the screen when the ALT or SHIFT / SHIFT-LOCK functionality on the keypad is used. When the icons are enabled, they overlay what is in the foreground LCD context, without altering the contents of the application's display context. When ALT or SHIFT is cancelled, the area of the icon returns to what the application has stored in the display context. By default, the icons are enabled. Icons can be enabled and disabled on a per-context basis. For example, see <code>bouncer.c</code> .

LcdPutStringXY

Puts text on the LCD display.

```
int LcdPutStringXY(
    int x,
    int y,
    const char * s,
    int length,
    int flags)
```

Parameters	x	The horizontal position in pixels from the left edge of the text window (if one has been defined) or the left edge of the display. x = -1 is treated specially as described below in length.
	y	The vertical position in pixels from the top edge of the text window (if one has been defined) or the top edge of the display. y = -1 means to use the current cursor's y position instead.
	s	A pointer to a string that is to be displayed at the position specified by the x and y parameters.
	length	The number of characters to be displayed. A length of -1 uses the current length of the string. However, if an actual length is specified, the string is not required to be NULL-terminated. If the actual string is longer than the value specified, the extra characters are not displayed. Fewer characters than specified might be displayed if, for example, line wrapping is disabled but the string is too long to be displayed on the current line.
	flags	Any of the following attributes, combined using the bitwise-OR operator:

Attribute	Description
TEXT_NORMAL	This is the default text attribute.
TEXT_UNDERLINE	Text is underlined.
TEXT_INVERT	Text is highlighted.
TEXT_OVERSTRIKE	Text is displayed without erasing what was already on the display at that location.
TEXT_BLANK	A display area of the correct height and width is erased, but the text is not displayed. This is equivalent to displaying the text in white on a white background.
LCD_RAW	This attribute prevents recognizing return characters and new lines as special characters.

Attribute	Description
LCD_LEFT_JUSTIFIED	Text is displayed starting at (x, y). This is the default behavior. If (x, y) is (-1, -1), the text starts at the current cursor position as set by the previous call to <code>LcdPutStringXY</code> .
LCD_RIGHT_JUSTIFIED	Text is displayed so that it ends at (x, y).
LCD_CENTERED	Text is displayed centered around (x, y). If x is -1, the text is horizontally centered within the text window.
LCD_MULTILINE	If this attribute is set and the string is too long to be displayed on the current line, the remaining characters are displayed on the next line. Otherwise, only those characters which can fit on the current line is displayed.
LCD_WORD_WRAP	This attribute prevents the display of partial words on a line. If the text is going to be truncated or wrapped to the next line, the wrap point will be on a word boundary. This cannot be combined with <code>lcd_right_justified</code> or <code>lcd_centered</code> .
LCD_ELLIPSIS	If text had to be truncated because of insufficient space on the line (if <code>lcd_multiline</code> is not used) or insufficient space on the screen (if <code>lcd_multiline</code> is used), then this attribute causes the last displayed character to be replaced by an ellipsis character (...) – character 0x1F, <code>char_ellipsis</code> .

Returns The actual number of characters processed. If there was not enough room to display all of the characters, this might be less than the `length` parameter.

Description A call to this function changes the current cursor position, so that a subsequent call with `x = -1, y = -1` and `flags = LCD_LEFT_JUSTIFIED` continues writing immediately after the string which was just displayed.

The following sample code demonstrates how to use this function.

```
LcdConfig lcdConf;
LcdGetConfig (&lcdConf);
LcdPutStringXY (-1, 0, "Underline", -1, TEXT_UNDERLINE | TEXT_CENTERED);
LcdPutStringXY(0, lcdConf.height/3, "Left-Justified", -1,
    TEXT_NORMAL | LCD_LEFT_JUSTIFIED);
LcdPutStringXY(-1, LCDCONF.HEIGHT*2/3, "Inverted", -1, TEXT_INVERT |
    TEXT_CENTERED);
```



Result of LcdPutStringXY sample code

See also `hello.c`, `bouncer.c`, `comtest.c`, and `ping.c`.

LcdRasterOp

Takes a source rectangular block out of a `BitMap` and applies a binary function `wOp` to it and the destination rectangular block in another bitmap.

```
void LcdRasterOp(  
    DWORD wOp,  
    DWORD wWide,  
    DWORD wHigh,  
    const BitMap * src,  
    int SrcX,  
    int SrcY,  
    BitMap * dest,  
    int DestX,  
    int DestY)
```

Parameters	<code>wOp</code>	This parameter has one of 16 possible functions which can be applied to each pixel. Refer to the following table for a description of the corresponding Windows - <code>BitBlt</code> Raster Operation Codes.
	<code>wWide</code>	The width of the source and destination block. The source and destination blocks are forced to be the same size accordingly.
	<code>wHigh</code>	The height of the source and destination block. The source and destination blocks are forced to be the same size accordingly.
	<code>src</code>	A pointer to the source bitmap structure. If <code>NULL</code> , then the application's current virtual display is used as the bitmap.
	<code>SrcX</code>	The horizontal location of the upper left corner of the source block, relative to the left edge of the display.
	<code>SrcY</code>	The vertical location of the upper left corner of the source block, relative to the top of the display.

- dest** A pointer to the destination bitmap structure. If NULL, then the application's current virtual display is used as the bitmap.
- DestX** The horizontal location of the upper left corner of the destination block, relative to the left edge of the display.
- DestY** The vertical location of the upper left corner of the destination block, relative to the top of the display.

The following table lists the Windows `BitBlt` Raster Operation Codes.

Operation code	Description
BLACK	This operation fills the destination area, ignoring the source bitmap. (BLACKNESS)
COPY_INVERT_SRC	This operation inverts the source bitmap and then copies it to the destination area. (NOTSRCCOPY)
COPY_SRC	This operation copies the source bitmap to the destination area. (SRCCOPY)
INVERT_DEST	This operation inverts the destination area, ignoring the source bitmap. (DSTINVERT)
INVERT_OF_SRC_AND_DEST	This operation logically exclusive AND-s each pixel of the inverted source bitmap with that of the destination bitmap and then places the result in the destination area.
INVERT_OF_SRC_OR_DEST	This operation inverts the source bitmap and logically OR-s each pixel with that of the destination bitmap and then places the result in the destination area.
INVERT_OF_SRC_XOR_DEST	This operation logically exclusive OR-s each pixel of the inverted source bitmap with that of the destination bitmap and then places the result in the destination area.
INVERT_SRC_AND_DEST	This operation inverts the source bitmap and logically AND-s each pixel with that of the destination bitmap and then places the result in the destination area.
INVERT_SRC_OR_DEST	This operation logically exclusive OR-s each pixel of the inverted source bitmap with that of the destination bitmap and then places the result in the destination area.
NO_OP	This operation has no effect on the destination area.

Chapter 6: LCD API

Operation code	Description
SRC_AND_DEST	This operation logically AND-s each pixel of the source bitmap with that of the destination bitmap and then places the result in the destination area. (SRCAND)
SRC_AND_INVERT_DEST	This operation logically AND-s each pixel of the source bitmap with that of the inverted destination bitmap and then places the result in the destination area.
SRC_OR_DEST	This operation logically exclusive OR-s each pixel of the source bitmap with that of the destination bitmap and then places the result in the destination area.
SRC_OR_INVERT_DEST	This operation logically exclusive OR-s each pixel of the source bitmap with that of the inverted destination bitmap and then places the result in the destination area. (MERGEPAIN)
SRC_XOR_DEST	This operation logically exclusive OR-s each pixel of the source bitmap with that of the destination bitmap and then places the result in the destination area. (SRCINVERT)
WHITE	This operation clears the destination area, ignoring the source bitmap. (WHITENESS)

Description This is an implementation of the standard computer graphics rectangular bitmap copy operation `rasterop`. It is also known as `bitblt` (BIT-wise BlockTransfer). This function takes a source rectangular block out of a `BitMap` (see `pager.h` for the typedef structure definition), and applies a binary function `wOp` to it and the destination rectangular block in another `BitMap`. There are 16 possible `wOp` functions which can be applied, as listed above.

`LcdRasterOp` does handle clipping properly: if you specify a source or destination block which extends past the edge of the source bitmap's boundaries, `LcdRasterOp` truncates the block.

For example, see `ping.c`.

LcdReplaceFont

Redefines a font, specified by `iFontIndex`, with a new font definition pointed to by `pNewFont`.

```
int LcdReplaceFont(
    int iFontIndex,
    const FontDefinition * pNewFont)
```

Parameters

<code>iFontIndex</code>	The index number of the font that is to be redefined. Valid numbers range from 0 to 4.
<code>pNewFont</code>	A pointer to a new font definition, or NULL.

Returns LCD_OK if successful; or LCD_BAD_FONT_INDEX if an invalid font number is specified.

Description Custom font definitions structures (in the form of C header files) are created by the `lcdfonts.exe` utility and included in the application. The user should ignore the details of the FONTDEFINITION structure and simply pass the FONT structure name as `pNewFont`. Refer to the *Developer Guide* for more information on using the `lcdfonts.exe` utility.

Specifying NULL as the font definition causes the system default font to be restored. The following table lists system default fonts:

Index	Font
0 (FONT_8_PIXEL)	An 8-pixel high font, allowing 8 lines of text on the display
1 (FONT_10_PIXEL)	A 10-pixel high font, allowing 6 lines of text on the display
2 (FONT_USER) to 4 (FONT_SET_MAX-1)	These indexes are reserved for user fonts. The default is the same as FONT_8_PIXEL.

LcdScroll

Scrolls the region defined by the current text window up or down.

```
void LcdScroll(int pixels)
```

Parameters

<code>pixels</code>	The number of pixels to scroll the current text window up (if positive) or down (if negative).
---------------------	--

Description This function scrolls the region defined by the current text window up or down. If no text window has been defined, it scrolls the entire display. If `pixels` is positive, the region is scrolled up, and an area at the bottom is cleared. If `pixels` is negative, the region is scrolled down, and an area at the top is cleared.

For example, see `comtest.c`, `hello.c`, and `ping.c`.

LcdSetContrast

Sets a new contrast setting for the LCD.

```
void LcdSetContrast(int contrast)
```

Parameters `contrast` The new contrast setting of the LCD; this value is a number from 0 (lightest) to the maximum contrast (darkest) setting support, which is `LCD_MAX_CONTRAST`.

For example, see `bouncer.c`.

LcdSetCurrentFont

Selects a new font as the current font.

```
int LcdSetCurrentFont(int iFontIndex)
```

Parameters `iFontIndex` The index number of the font that will be used for future text display; valid numbers range from 0 to 4.

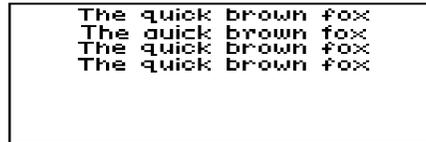
Returns 0 (zero) if the font selection was successful; otherwise, it returns `LCD_BAD_FONT_INDEX` if the font index number is invalid.

Description This function selects a new font as the current font. Subsequent text is displayed using the new font.

The following sample code demonstrates how to use this function:

```
// Select font #2, which is known to be a valid font.
LcdSetCurrentFont(2);

// Select font #number
int number;
if (LcdSetCurrentFont(number) != 0) {
    LcdClearDisplay();
    LcdPutStringXY(-1, 0, "Font selection error", -1, LCD_CENTERED);
}
// Print text in a variety of fonts
for (x=0; x<=3; x++) {
    LcdSetCurrentFont(x);
    LcdPutStringXY(-1, x*8, "The quick brown fox", -1, LCD_CENTERED);
}
```



```
The quick brown fox
The quick brown fox
The quick brown fox
The quick brown fox
```

Result of LcdSetCurrentFont sample code

LcdSetCursorPoint

Moves the cursor to a new pixel position defined by `xPixel` and `yPixel`.

```
void LcdSetCursorPoint(
    int xPixel,
    int yPixel)
```

Parameters

- `xPixel` The new horizontal cursor position (0=left).
- `yPixel` The new vertical cursor position (0=top).

Description This function moves the cursor to a new pixel position defined by `xPixel` (horizontal from left to right) and `yPixel` (vertical from top to bottom). If a text window has been defined, the cursor is positioned relative to the text window's (not the display's) top left-hand corner.

The following sample code demonstrates how to use this function:

```
// Move the cursor to pixel row 2, pixel column 13, and display text.
int x, y;
LcdSetCursorPoint(13, 2);

LcdGetCursorPoint(&x, &y);
LcdPutStringXY(x, y, "Hello World", -1, TEXT_NORMAL);
```



```
Hello World
```

Result of LcdSetCursorPoint sample code

See also `regmess.c`.

LcdsetDisplayContext

Activates the display context numbered `iDC`.

```
int LcdsetDisplayContext(int iDC)
```

Chapter 6: LCD API

Parameters `iDC` The number of the display context.

Returns `LCD_OK` if successful, or `LCD_BAD_HANDLE` if an invalid display context is chosen.

Description This function activates the display context numbered `iDC`. A display context is a buffer which can be accessed just like the regular display, but remains invisible until it is activated. This is useful for leaving the previous display on until the new display is ready, as well as for task-swapping.

When the display context, `SYSTEM_MODAL_DC`, is selected the current screen image is frozen and the program requesting the `SYSTEM_MODAL_DC` can draw on top of it. This does not bring the calling application to the foreground. The current foreground program continues to run, to receive keystroke messages, and to update its (now hidden) display context. When the task which had selected `SYSTEM_MODAL_DC` selects a different display context, the screen is refreshed from the foreground program's display context, and normal screen updates resume.

For example, see `bouncer.c`.

LcdSetPixel

Provides a very low-level interface to the LCD display system.

```
void LcdSetPixel(  
    int x,  
    int y,  
    BOOL value)
```

Parameters `x` The horizontal pixel position (0=left).

`y` The vertical pixel position (0=top).

`value` True to turn the LCD pixel at the specified (`x`, `y`) on and false to turn it off.

LcdSetRegion

Copies contents of a user buffer to a region of the LCD.

```
void LcdSetRegion(  
    int left,  
    int top,  
    int width,  
    int height,  
    void * BitArray)
```

Parameters	<code>left</code>	The horizontal pixel location of the top left-hand corner of the rectangular area on the LCD.
	<code>top</code>	The vertical pixel location of the top left-hand corner of the rectangular area on the LCD.
	<code>width</code>	The pixel width of the rectangular area on the LCD.
	<code>height</code>	The pixel height of the rectangular area on the LCD.
	<code>BitArray</code>	The sequence of bits, packed 8 to a byte, to be sent to the rectangular area on the LCD. The least significant bit (LSB) is first.

Description This function is equivalent to the following example code:

```
for (x=left; x <left+width; x++){
for (y=top; y <top+height; y++) {
LcdSetPixel (x, y, next bit from BitArray);
}
// skip bits in BitArray if necessary to get to a byte boundary
}
```

LcdSetTextWindow

Defines the boundaries of the text window.

```
void LcdSetTextWindow(
    int x,
    int y,
    int wide,
    int high)
```

Parameters	<code>x</code>	The horizontal pixel location of the top left corner of the new text window.
	<code>y</code>	The vertical pixel location of the top left corner of the new text window.
	<code>wide</code>	The pixel width of the new text window.
	<code>high</code>	The pixel height of the new text window.

Description This function defines the boundaries of the text window. Subsequent text display will be confined within this window. This function is useful for keeping certain information in one place on the display, while placing new text in a different location.

The following sample code demonstrates how to use this function:

```
// Fill the screen with the letter "A", then define a text window, clear
// it, and write the text "295 Phillip" in the new text window.
```

Chapter 6: LCD API

```

int charWidth,charHeight;
int fontIndex = LcdGetCurrentFont();
charWidth = LcdGetCharacterWidth('A', fontIndex);
charHeight = LcdGetFontHeight(fontIndex);
for (y=0; y<(65/charHeight); y++)
    for (x=0; x<(132/charWidth); x++)
        LcdPutStringXY(x*charWidth, y*charHeight, "A", 1, TEXT_NORMAL);
LcdSetTextWindow(20, 12, 80, 25);
LcdClearDisplay();
LcdPutStrinXY(0, 0, "295 Phillip", -1, TEXT_NORMAL);

```



Result of LcdSetCurentText sample

Constants

A large number of very specific constants are not reproduced here. For information about RasterOperations and TextFormatting, please see LcdRasterOp and LcdPutStringXY, respectively.



Note: The constants LCD_WIDTH, LCD_HEIGHT, LCD_HEIGHT_BYTE, LCD_DISPLAY_SIZE, and LCD_DISPLAY_SIZE_BYTE are no longer available. The information they provided should be obtained using the LcdGetConfig function.

Constant	Description
LCD_MAXCONTRAST	Maximum contrast index. The valid range of contrast settings is [0, LCD_MAXCONTRAST].
DRAW_WHITE	This drawing mode specifies that objects placed on the display should be drawn by turning off pixels.
DRAW_INVERT	This drawing mode specifies that objects placed on the display should be drawn by turning on pixels that are off and turning off pixels that are on.
DRAW_BLACK	This drawing mode specifies that objects placed on the display should be drawn by turning on pixels.
FONT_8_PIXEL	This font index specifies the standard 8 pixel high font. In this font, 8 rows of characters are placed on the display.

Return codes

Constant	Description
FONT_10_PIXEL	This font index specifies the standard 10 pixel high font. In this font, 6 rows of characters are placed on the display.
FONT_USER	This font index specifies the first user defined font.
FONT_SET_MAX	This font index specifies one more than the maximum user font index. The valid user font index range is font_user to (font_set_max - 1).
FONT_REGULAR	This font index refers to the regular system font. This is currently an alias for font_8_pixel.

Return codes

Return Code	Description
LCD_OK	Indicates that the operation was successful
LCD_BAD_HANDLE	Returned when an invalid display context is specified
LCD_OUT_OF_MEMORY	Returned if the operation failed due to a lack of system resources
LCD_BAD_FONT_INDEX	Returned if the specified font index is invalid

Chapter 6: LCD API

Chapter 7

Peripheral API

The RIM 857™ and RIM 957™ can host peripheral devices, such as bar code readers and credit card swipers. Most peripherals attach to and communicate via the serial port.

The Peripheral API provides access to signals and control lines available on the RIM 857 and RIM 957 beyond the serial interface.

When the handheld detects a peripheral connection, it sends a PERIPHERAL_ID_UPDATE event to applications. When the controlling application receives this event, it should call `PeripheralGetId()` to determine whether the event indicates a peripheral connection or a peripheral disconnection (an ID of 4 indicates that a peripheral is connected). The application can then call `PeripheralRegister()` to obtain ownership of the peripheral port.

Functions

The following Peripheral API functions, declared in `peripheral.h`, are listed alphabetically.

<code>PeripheralGetId</code>	108
<code>PeripheralGetOwner</code>	108
<code>PeripheralPowerSupply</code>	108
<code>PeripheralRegister</code>	108
<code>PeripheralStatusControl</code>	109
<code>PeripheralUnregister</code>	109

PeripheralGetId

Retrieves the current status of the PER_IDx pins.

```
int PeripheralGetId(void);
```

Returns Bit x of the return value corresponds to PER_IDx. For example, a return value of 4 means that (PER_ID2, PER_ID1, PER_ID0) equals (1, 0, 0) and indicates that a peripheral is connected.

PeripheralGetOwner

Retrieves the ID of the task that currently has ownership of the peripheral port.

```
TASK PeripheralGetOwner(void)
```

Returns The ID of the task that currently has ownership of the peripheral port, or TASK_NOT_FOUND if no task currently owns the port.

PeripheralPowerSupply

Retrieves, and optionally changes, the status of the power supply pin.

```
int PeripheralPowerSupply(int iMode)
```

Parameters

iMode	Status of the power supply; one of the following values: PER_POWER_ON PER_POWER_OFF PER_POWER_QUERY
-------	--

Returns This function returns the current status of the peripheral power supply, or -1 if the calling task does not currently have ownership of the peripheral port.

Description The current status of the power supply pin, and then sets it to the status specified by iMode. If iMode is PER_POWER_QUERY, the function just returns the current status and makes no change.

PeripheralRegister

Registers ownership of the peripheral port with current task.

```
BOOL PeripheralRegister()
```

Returns True if ownership of the peripheral port is assigned successfully to the calling task; false otherwise.

Description This function gives the calling task ownership of the peripheral port if a peripheral is attached, and no other task currently owns it. A task must have ownership of the peripheral port to control the power supply and status/control pins.

Refer to "PeripheralUnregister" on page 109 for more information.

PeripheralStatusControl

Retrieves, and optionally changes, the current status of the status/control data latch.

```
int PeripheralStatusControl(int iMode)
```

Parameters `iMode` Status of the power supply; one of the following values:
 `PER_SC_QUERY_LATCH`
 `PER_SC_QUERY_STATUS`
 `PER_SC_LATCH_OFF`
 `PER_SC_LATCH_ON`

Returns The value in the status/control data latch (if `iMode` is set to `PER_SC_QUERY_LATCH`) or returns the input state of the status/control pin and sets the latch to the state specified in the `iMode` parameter.

The function returns -1 if the calling task does not currently have ownership of the peripheral port.

PeripheralUnregister

Relinquishes ownership of the peripheral port by the calling task.

```
BOOL PeripheralUnregister()
```

Returns True if the calling task successfully relinquishes ownership of the peripheral port; false if the task did not already have ownership of the port.

Description This function relinquishes ownership of the peripheral port.
 Refer to "PeripheralRegister" on page 108 for more information.

Chapter 7: Peripheral API

Chapter 8

Device events

This chapter lists the types of device events. When a device event occurs, the `MESSAGE.Devi ce` member is equal to one of these values:

- `DEVICE_SYSTEM`—system device events (refer to page 112)
- `DEVICE_TIMER`—timer device events (refer to page 114)
- `DEVICE_RTS`—real-time clock events (refer to page 114)
- `DEVICE_HOLSTER`—holster events (refer to page 115)
- `DEVICE_KEYPAD`—input events (refer to page 115)
- `DEVICE_RADIO`—radio events (refer to page 118)
- `DEVICE_COM`—serial port events (refer to page 119)

SYSTEM device events

When the following events occur, the Device member of the MESSAGE structure will be equal to DEVICE_SYSTEM.

Event	Description
BATTERY_GOOD	The internal battery is back to normal. Sent after recharging.
BATTERY_LOW	When the internal battery voltage falls below a critical level, this message is sent to all applications. The applications should then save any necessary information to the flash file system. To start recharging the battery, the user must plug in the external power adapter or install new AA batteries.
BATTERY_UPDATE	Application should call <code>RimGetBatteryLevel()</code> to get the latest state.
MEMORY_LOW	Sent when memory drops below a specific threshold; a request to free up RAM.
PERIPHERAL_ID_UPDATE	Application should call <code>PeripheralGetId()</code> to get the latest ID.
POWER_OFF	This event is sent to all applications to indicate that the user is putting the device into a power off state. Although power is not actually off, the applications do not receive any messages, the LCD display is blank in power-save mode, and the radio is off. An application will turn the power off by calling <code>RimRequestPowerOff</code> . The COM port, if open at power down, will be closed automatically.
POWER_UP	This event is sent to all applications to indicate that the device has left the power off state. This may be caused by a key or trackwheel event, a real-time clock alarm event, or a change of state on the COM control lines indicating that a serial cable has been plugged in.

Event	Description
SWITCH_BACKGROUND	This event is sent to the current foreground application, indicating that a background application has requested foreground. It indicates that the foreground application has been forced to the background. The SubMsg field contains the task handle of the application that requested foreground.
SWITCH_FOREGROUND	This event is sent to a background application to indicate that it has received foreground.
TASK_LIST_CHANGED	This message is sent to the foreground task when another task has changed its task switcher information by a call to <code>RimSetPID</code> .

TIMER device events

When the following events occur, the Device member of the MESSAGE structure will be equal to DEVICE_TIMER.

The handheld contains a pool of 48 global timers, for which applications can register, using the RimSetTimer function. These timers can be configured as either periodic or one-time, with a minimum resolution of 10 milliseconds, and trigger and event when the timer has expired.

To register a timer, pass a timer ID, the length of the timer (in 1/100 second increments), and the type of the timer, either TIMER_PERIODIC, TIMER_ONE_SHOT, TIMER_ABSOLUTE to the RimSetTimer function.

When a timer expires, a message from DEVICE_TIMER is sent to the application that called RimSetTimer. The Event field of this message will contain the timer ID assigned by the application when it called RimSetTimer.

In order to avoid overflowing the system with events, newer messages of periodic timers will not be dispatched until the previous message is received by the application. As such, there is potential for drift in periodic timers over time if the system is very busy.

In order to prolong battery life, avoid using short periodic timers (less than 5 seconds) for extended periods.

RTC (real-time clock) device events

When the following events occur, the Device member of the MESSAGE structure will be equal to DEVICE_RTC.

Event	Description
RTC_ALARM_EXPIRED	The real-time clock device sends this event to indicate that the alarm time has been reached. This event is sent only to the application that has set the alarm clock to the current time. Every application can set a different alarm time. Also see the RimSetAlarmClock API call for information on setting the alarm time.
RTC_CLOCK_UPDATE	This event is sent to all applications whenever the date and time is updated. This occurs every minute or when the date and time has been programmed. Applications can then call RimGetDateTime to get the current date and time.

HOLSTER device events

When the following events occur, the Device member of the MESSAGE structure will be equal to DEVICE_HOLSTER.

Event	Description
IN_HOLSTER	This event is sent to all applications whenever the device is put into the holster.
OUT_OF_HOLSTER	This event is sent to all applications whenever the device is removed from the holster. Note: See the function <code>RimHolsterStatus</code> for information on how to get the current holster status. Neither of these messages is sent to the applications if the device is in Power Off mode.

KEYPAD device events

When the following events occur, the Device member of the MESSAGE structure will be equal to DEVICE_KEYPAD.

The `SubMsg` field contains the ASCII code of the key that is pressed, including any effects of the ALT or SHIFT keys. The status of the ALT, SHIFT, and CAPS LOCK keys can be obtained from the `Data[0]` field as a combination of the `ALT_STATUS`, `SHIFT_STATUS` and `CAPS_LOCK` constants. The `Data[1]` field indicates the time at which the event happened as the number of 10 millisecond increments since the device was turned on.

Event	Description
THUMB_CLICK	The trackwheel has been pressed down.
THUMB_UNCLICK	The trackwheel has been released.
THUMB_ROLL_UP	The trackwheel has been rotated up.
THUMB_ROLL_DOWN	The trackwheel has been rotated down. The amount by which the trackwheel has rotated since the last THUMB_ message was sent is placed in the <code>SubMsg</code> field of the message.

Chapter 8: Device events

Event	Description
KEY_DOWN	The key has been pressed down.
KEY_REPEAT	The key has been held down and is automatically repeating. The key_repeat event is generated only when the API function KeypadRate is used to enable repeating keys.
KEY_STATUS	The ALT or SHIFT status has changed. The ALT and SHIFT keys are not passed to an application in the normal way. Instead, they are processed by the application server.

The following table shows the possible values in the SubMsg field for KEY_DOWN and KEY_REPEAT events. For all key presses, the SubMsg field will contain the ASCII value of the character shown.

Key	SHIFT + Key	ALT + Key
a	A	* <i>asterisk</i>
b	B	! <i>exclamation mark</i>
c	C	; <i>semicolon</i>
d	D	+ <i>plus sign</i>
e	E	3
f	F	- <i>minus, hyphen</i>
g	G	= <i>equal sign</i>
h	H	: <i>colon</i>
l	I	8
j	J	' <i>apostrophe</i>
k	K	" <i>double quotation mark</i>
l	L	@ <i>at sign</i>
m	M	. <i>period</i>
n	N	, <i>comma</i>
o	O	9

Key	SHIFT + Key	ALT + Key
p	P	0
q	Q	1
r	R	4
s	S	<i>/ forward slash</i>
t	T	5
u	U	7
v	V	<i>? question mark</i>
w	W	2
x	X	<i>\$ dollar sign</i>
y	Y	6
z	Z	<i>_ underscore</i>
Space bar	space bar	space bar
ENTER	ENTER	ENTER
BACKSPACE	SHIFT_BACKSPACE	BACKSPACE

RADIO device events

When the following events occur, the `Device` member of the `MESSAGE` structure will be equal to `DEVICE_RADIO`. The following descriptions are generic; descriptions specific to your wireless network can be found in the RADIO device events section of the *Radio API Developer Guide*.

Event	Description
BASE_STATION_CHANGE	This event is sent to all registered applications. It indicates that the radio modem has changed base stations.
MESSAGE_NOT_SENT	An attempt to transmit information to the network failed. This event is sent to the task who submitted the packet whenever coverage is too poor for transmission or an invalid data packet is sent.
MESSAGE_RECEIVED	This event is sent to all applications that have registered to receive RADIO events (<code>RadioRegister()</code>). This event indicates that a data packet was received. Applications should call the appropriate <code>RadioGet()</code> function to receive the message data.
MESSAGE_SENT	This event is an acknowledgement that a transmitted packet was received by the wireless network. This event is sent to the application that sent the packet, whether that application is in the foreground or the background. The <code>SubMsg</code> field contains the tag value that was returned by the appropriate <code>RadioSend</code> call.
MESSAGE_STATUS	This event notifies the sender of the packet's transmit status. A data packet sent to the Radio API may not be transmitted immediately, because the packets are queued in the Radio API layer, and the radio code's retry spacing. The <code>Data[0]</code> field of the <code>MESSAGE</code> structure will contain the status subcodes.
NETWORK_STARTED	This event, sent to all registered applications, indicates that the radio modem has been turned on or has just switched to a new network.
RADIO_TURNED_OFF	This event is sent to all registered applications, indicating that the radio modem has been turned off, either by the user or as a result of a low battery.
SIGNAL_LEVEL	This event is sent to all registered applications, and indicates that the receive signal level has changed. The <code>SubMsg</code> field contains a negative value, which represents the level of the signal in dBm. A less negative value (closer to zero) indicates a stronger signal. A value of <code>-256 dBm</code> (<code>RSSI_NO_COVERAGE</code>) indicates that the modem is out of coverage.

COM device events

When the following events occur, the Device member of the MESSAGE structure will be equal to DEVICE_COM1.



Note: Events are always sent to the task that has opened the communications port.

The COM driver does not support flow control. If any form of flow control, such as XON and XOFF is required, it is the application's responsibility to monitor the status of flow control, and whether to send data or not at appropriate times.

Event	Description
COMM_CONTROL_CHANGE	This event indicates a level change on the DTR serial port input. The new state of the control line is placed in the SubMsg field.
COMM_PATTERN_NOTIFY	This event indicates that the pattern registered with <code>CommRegisterNotifyPattern()</code> has been detected on the serial port while no application has it opened. Note: This message is only generated when no application has opened the serial port.
COMM_RX_AVAILABLE	This event indicates that the serial receive queue has changed from empty to not empty. Only one COMM_RX_AVAILABLE message is sent for each time that the serial buffer changes from empty to not empty. An application will not receive another COMM_RX_AVAILABLE message until it has completely emptied the serial buffer and new characters are received. See <code>CommReadBuffer()</code> for more details.
COMM_RX_ERROR	If a communications error, such as receive overrun, framing error, or a parity error is received, a COMM_RX_ERROR message is sent to the task that opened the COM port. The SubMsg field of the message will contain the specific error number.
COMM_TX_EMPTY	A COMM_TX_EMPTY event is sent to the task that opened the COM port whenever the port's transmit FIFO, as well as the serial transmit hardware, becomes completely empty.
POWER_OFF	When the device is powered off while an application has the COM port open, the COM port is automatically closed. If the application wishes to resume use of the COM port when the device is turned back on, it must open the port again. POWER_OFF is not a COM-specific event, but applications making use of the COM port must be aware of the effects of being powered off.

Chapter 8: Device events

Index of functions

F

File system

- DbAddOrphan(), 49
- DbAddRec(), 49
- DbAndRec(), 51
- DbDelete(), 52
- DbDeleteRec(), 52
- DbFileClose(), 53
- DbFileInfo(), 54
- DbFileOpen(), 54
- DbFileRead(), 55
- DbFileSeek(), 55
- DbFileSysInfo(), 56
- DbFileWrite(), 57
- DbFindNext(), 58
- DbFirstRec(), 59
- DbFreeRec(), 59
- DbFreeSpace(), 59
- DbGetHandle(), 60
- DbMaxHandles(), 61
- DbMaxNewRecSize(), 61
- DbName(), 62
- DbNextRec(), 62
- DbPointTable(), 63
- DbPointTableEdition(), 63
- DbRecSize(), 63
- DbReplaceOrphan(), 64
- DbReplaceRec(), 64
- DbSecure(), 65
- DbSize(), 65

K

Keypad

- KeypadBeep(), 77
- KeypadRate(), 77
- KeypadRegister(), 78

L

LCD

- LcdClearDisplay(), 82
- LcdClearToEndOfLine(), 83
- LcdCopyBitmapToDisplay(), 83

- LcdCopyDisplay(), 84
- LcdCreateDisplayContext(), 84
- LcdDestroyDisplayContext(), 85
- LcdDrawBox(), 85
- LcdDrawLine(), 86
- LcdForceRefresh(), 87
- LcdGetCharacterWidth(), 87
- LcdGetConfig(), 88
- LcdGetContrast(), 88
- LcdGetCurrentFont(), 88
- LcdGetCursorPoint(), 88
- LcdGetDisplayContext(), 89
- LcdGetFontHeight(), 89
- LcdGetFontName(), 91
- LcdGetNumberOfFonts(), 91
- LcdGetPixel(), 92
- LcdGetRegion(), 92
- LcdGetStringWidth(), 93
- LcdIconsEnable(), 93
- LcdPutStringXY(), 94
- LcdRasterOp(), 96
- LcdReplaceFont(), 99
- LcdScroll(), 99
- LcdSetContrast(), 100
- LcdSetCurrentFont(), 100
- LcdSetCursorPoint(), 101
- LcdSetDisplayContext(), 101
- LcdSetPixel(), 102
- LcdSetRegion(), 102
- LcdSetTextWindow(), 103

M

Memory

- RimFree(), 24
- RimGetMaxAllocSize(), 24
- RimMalloc(), 24
- RimMemoryRemaining(), 25
- RimRealloc(), 25

P

palm-sized device

- RimConfigureLEDs(), 35, 43

Index of functions

- RimSetLed(), 44
- Password
 - RimPasswordFailureCount(), 34
 - RimSetPassword(), 33
 - RimVerifyPassword(), 33
- Peripheral
 - PeripheralGetId(), 108
 - PeripheralGetOwner(), 108
 - PeripheralPowerSupply(), 108
 - PeripheralRegister(), 108
 - PeripheralStatusControl(), 109
 - PeripheralUnregister(), 109
- S**
- Serial
 - CommClosePort(), 70
 - CommGetDtr(), 70
 - CommOpenPort(), 70
 - CommReadBuffer(), 71
 - CommReadChar(), 71
 - CommRegisterNotifyPattern(), 72
 - CommSendBuffer(), 73
 - CommSendChar(), 73
 - CommSetDsr(), 73
 - CommSetFlowControl(), 74
 - CommSettings(), 74
 - CommStandbyMode(), 74
 - CommTxCount(), 75
- String
 - RimDebugPrintf(), 36
 - RimSprintf(), 44
 - RimVsprintf(), 46
- System
 - RimAlertNotify(), 30
 - RimCatastrophicFailure(), 35
 - RimConfigureLEDs(), 35, 43
 - RimCreateThread(), 13
 - RimDisableAppSwitch(), 13
 - RimEnableAppSwitch(), 13
 - RimFindTask(), 14
 - RimGetAlarm(), 26
 - RimGetAlertConfiguration(), 31
 - RimGetBatteryLevel(), 36
 - RimGetBatteryStatus(), 37
 - RimGetCurrentTaskID(), 14
 - RimGetDeviceInfo(), 37
 - RimGetForegroundApp(), 14
 - RimGetLanguage(), 38
 - RimGetLoadedAppInfo(), 38
 - RimGetMessage(), 14
 - RimGetMessageRaw(), 39
 - RimGetNumberOfTunes(), 31
 - RimGetOSversion(), 40
 - RimGetPID(), 16
 - RimGetSetOfLanguages(), 41
 - RimGetTuneName(), 31
 - RimHolsterStatus(), 41
 - RimInitiateReset(), 42
 - RimInvokeTaskSwitcher(), 16
 - RimPeekMessage(), 16
 - RimPostMessage(), 17
 - RimPowerDownHandheld(), 42
 - RimRegisterForPowerDown(), 42
 - RimRegisterMessageCallback(), 17
 - RimReplyMessage(), 19
 - RimRequestForeground(), 20
 - RimRequestFullPowerOff(), 42
 - RimRequestPowerOff(), 42, 43
 - RimRequestStorageMode, 43
 - RimRequestStorageMode(), 43
 - RimSendMessage(), 20
 - RimSendSyncMessage(), 20
 - RimSetAlarmClock(), 28
 - RimSetAlertConfiguration(), 32
 - RimSetLanguage(), 43
 - RimSetLed(), 44
 - RimSetPID(), 21
 - RimSetReceiveFromDevice(), 21
 - RimSleep(), 30
 - RimSpeakerBeep(), 32
 - RimStackUsage(), 46
 - RimTaskYield(), 22
 - RimTerminateThread(), 22
 - RimTestAlert(), 32
 - RimToggleMessageReceiving(), 23
 - RimWaitForSpecificMessage(), 23
- T**
- Time
 - RimGetDateTime(), 26
 - RimGetIdleTime(), 27
 - RimGetTicks(), 27
 - RimKillTimer(), 27
 - RimSetDate(), 28
 - RimSetTime(), 28
 - RimSetTimer(), 29

Index

A

- alarm clock See real-time clock
- ALARM_EXPIRED See RTC_ALARM_EXPIRED
- API functions
 - keypad, 77
 - LCD, 82
 - serial communications, 69
- application server, 16, 21, 116
- application stack size, 13
- applications
 - reject incoming messages, 23
 - retrieving information, 38
 - stopping, 30
 - waiting for specific message, 23

B

- background application
 - bringing to foreground, 13, 20, 113
 - requesting handle, 14
- base station, 118
- BASE_STATION_CHANGE, 118
- battery, 43
- battery power
 - battery level, 36
 - low batteries, 112, 118
 - recharging the batteries, 112
 - tips for conserving power, 114
- BATTERY_GOOD, 112
- BATTERY_LOW, 112
- BATTERY_UPDATE, 112
- BATTERY_UPDATE event, 37
- bitblt See bitmaps, copying
- bitmaps
 - copying, 96
 - displaying, 83

C

- clearing display, 82, 83, 99
- CLOCK_UPDATE See RTC_CLOCK_UPDATE

COM port

- closing, 70
- configuration, 74
- get DTR state, 70
- opening, 70
- receive buffer, 71
- receiving characters, 71
- sending characters, 73
- setting DSR state, 73
- setting flow control, 74
- standby mode, 74
- transmit buffer, 73, 75

com port

- device events, 119

COMM_CONTROL_CHANGE, 119

COMM_PATTERN_NOTIFY, 119

COMM_PATTERN_NOTIFY event, 72

COMM_RX_AVAILABLE, 119

COMM_RX_ERROR, 119

COMM_TX_EMPTY, 119

contrast, 88, 100

cursor

- getting position, 88, 92
- moving, 101

customizing country-dependent behaviour, 38, 43

customizing language-dependent behaviour, 38, 43

D

- data packet, 118
- database
 - creating, 60
- database / file system
 - adding record, 49
 - attaching record, 49
 - closing file, 53
 - creating database, 60
 - database handle, 56, 60, 62
 - database name, 56, 62
 - database size, 52, 65
 - deleting record, 52

Index

- error codes, 67
- file status information, 54
- finding the first record, 59
- finding the next record, 62
- free space, 59
- maximum file handles, 61
- maximum new record size, 61
- maximum record handles, 61
- modifying an orphan, 64
- modifying record, 51, 64
- opening file, 54
- pattern matching, 58
- reading, 55
- record handles, 59
- record pointer table, 63
- record size, 63
- secure database, 65
- setting current position, 55
- writing data to file, 57

Database / file system error codes, 67

database name

- maximum size, 60

device events, 111

- com, 119
- holster, 115
- keypad, 115
- radio, 118
- rtc (real-time clock), 114
- system, 112
- timer, 114

display

- enable and disable icons, 93

display context

- copying, 84
- copying bitmap to, 83
- creating, 84
- defined, 84, 89
- deleting, 85
- getting handle, 89
- setting, 101

displaying

- graphics, 83, 85, 86
- text, 94, 103

DSR, 73

DTR, 70, 72

dtr, 119

E

events

- BASE_STATION_CHANGE, 118
- BATTERY_GOOD, 112
- BATTERY_LOW, 112

- BATTERY_UPDATE>, 112
- COMM_CONTROL_CHANGE, 119
- COMM_PATTERN_NOTIFY, 119
- COMM_RX_AVAILABLE, 119
- COMM_TX_EMPTY, 119
- IN_HOLSTER, 115
- INITIALIZE, 14
- KEY_DOWN, 116
- KEY_REPEAT, 116
- KEY_STATUS, 116
- MEMORY_LOW, 112
- MESSAGE_NOT_SENT, 118
- MESSAGE_SENT, 118
- MESSAGE_STATUS, 118
- NETWORK_STARTED, 118
- notification, 20, 28
- OUT_OF_HOLSTER, 115
- PERIPHERAL_ID_UPDATE, 112
- POWER_OFF, 112, 119
- POWER_UP, 14, 112
- queue, 16
- RADIO_TURNED_OFF, 118
- RTC_ALARM_EXPIRED, 114
- RTC_CLOCK_UPDATE, 114
- SIGNAL_LEVEL, 118
- SWITCH_BACKGROUND, 20, 113
- SWITCH_FOREGROUND, 20, 113
- TASK_LIST_CHANGED, 113
- THUMB_CLICK, 115
- THUMB_ROLL_DOWN, 115
- THUMB_ROLL_UP, 115
- THUMB_UNCLICK, 115

F

files

- creating streamed, 60
- streamed, creating, 60

flash file system, 112

fonts

- attributes, 84
- creating new font, 99
- getting name, 91
- getting number of, 91
- height of current font, 88, 89
- selecting new font, 100
- width of current font, 87

foreground application, 16

- requesting handle, 14
- sending to background, 13, 20, 113

formatting and printing characters to output stream, 36

formatting character strings, 44, 46

freeing memory, 24

G

getting
 amount of time handheld has been on, 27
 battery level, 36
 battery status, 37
 message, 14
 raw messages, 39
 getting device information, 37
 getting name of tune, 31
 getting number of tunes, 31

H

holster, 41
 device events, 115

I

idle time, 27
 IN_HOLSTER, 115
 INITIALIZE event, 14
 inter-process communications
 asynchronous (non-blocking) send, 17

K

KEY_DOWN, 116
 KEY_REPEAT, 116
 KEY_STATUS, 116
 keypad, 27
 configuring for auto key repeat, 77
 device events, 115
 functions, 77
 intercepting global hot keys, 78
 repeating keys, 78, 116

L

language
 retrieving available, 41
 retrieving current, 38
 setting current, 43

LCD

clearing display, 82, 83, 99
 copying bitmaps, 96
 copying displays, 84
 creating display context, 84
 creating new font, 99
 deleting display context, 85
 disabling icons, 93
 displaying bitmaps, 83
 displaying graphics, 85
 displaying text, 94, 103
 drawing lines, 86

enabling icons, 93
 functions, 82
 getting contrast, 88
 getting current font width, 87
 getting cursor position, 88, 92
 getting display context handle, 89
 getting font name, 91
 getting height of current font, 88, 89
 getting number of fonts, 91
 getting region, 92
 getting width of string, 93
 moving the cursor, 101
 refreshing, 87
 selecting new font, 100
 setting contrast, 100
 setting display context, 101
 setting region, 102
 turning off pixels, 102
 turning on pixels, 102

led

configure lighting, 35
 set state, 44

M

memory
 allocating, 24
 freeing, 24
 reallocating, 25
 remaining, 25
 MEMORY_LOW, 112
 message passing, 16, 17, 20, 22, 39, 45
 MESSAGE structure, 15, 38, 39
 MESSAGE_NOT_SENT, 118
 MESSAGE_SENT, 118
 MESSAGE_STATUS, 118
 messages
 replying, 19
 sending synchronous, 20
 modem out of coverage, 118
 multiple applications, 14, 72
 multi-tasking, 5

N

network coverage too poor for transmission, 118
 NETWORK_STARTED, 118
 notification, 30, 31, 32

O

operating system
 version, 40
 OUT_OF_HOLSTER, 115

Index

P

- pager.h, 98
- palm-sized wireless handheld, 35, 43, 44
- password
 - failures, 34
 - setting, 33
 - verifying, 33
- pattern matching, 58, 72
- PERIPHERAL_ID_UPDATE, 112
- peripherals
 - deregistering, 109
 - owner, 108
 - pin status, 108
 - power supply, 108
 - registering, 108
 - status control, 109
- PID, 16, 21
- pixel
 - turning on and off, 102
 - width of string, 93
- power
 - power down, 42, 43
 - registering for power down, 42
 - storage mode, 43
- power adapter, 112
- power save mode, 112
- POWER_OFF, 112, 119
- POWER_OFF event, 42, 43
- POWER_UP, 112
- POWER_UP event, 14

R

- radio device events, 118
- RADIO_TURNED_OFF, 118
- RasterOp See bitmaps, copying
- real-time clock
 - alarm clock, 115
 - device events See rtc, device events
 - next alarm, 26
 - reading the date and time, 26
 - setting the alarm clock, 28
 - updating date and time, 114
- receiving from devices, 21
- receiving packets, 118
- refreshing the LCD, 87
- repeating keys, 78, 116
- resetting, 42
- right justify, 94
- RimGetDateTime(), 114
- RimHolsterStatus(), 115
- RimSetTimer(), 114
- rtc device events, 114
- RTC_ALARM_EXPIRED, 114

- RTC_ALARM_EXPIRED events, 28
- RTC_CLOCK_UPDATE, 114

S

- scrolling, 99
- sending
 - data packets, 118
 - synchronous messages, 20
- serial communications
 - constants and error codes, 75
 - functions, 69
- Serial communications constants and error codes, 75
- setting the time on the handheld, 28
- signal strength, 118
- SIGNAL_LEVEL, 118
- simulator
 - restrictions, 28, 36
- storage mode, 43
- SubMsg field, 113, 115, 117, 118, 119
- SWITCH_BACKGROUND, 113
- SWITCH_BACKGROUND event, 20
- SWITCH_FOREGROUND, 113
- SWITCH_FOREGROUND event, 20
- system
 - allocating memory, 24
 - cancelling the timer, 27
 - choosing value for stack size, 46
 - creating threads, 13
 - customizing the language, 38
 - device events, 112
 - disabling switching of applications, 13
 - enabling switching of applications, 13
 - finding if handheld is in the holster, 41
 - finding task ID, 14
 - formatting characters, 44, 46
 - freeing memory, 24
 - generation of tones, 32
 - getting alarm, 26
 - getting battery level, 36
 - getting battery status, 37
 - getting configuration settings, 31
 - getting current task ID, 14
 - getting device information, 37
 - getting foreground application, 14
 - getting idle time, 27
 - getting name of tune, 31
 - getting next message, 14
 - getting number of tunes, 31
 - getting process attributes, 16
 - getting raw messages, 39
 - getting the date and time, 26
 - getting the OS version, 40
 - getting time handheld has been on, 27

- notifying the user, 30
- powering down, 42
- printing text to debug stream, 36
- processing messages, 16, 17
- reallocating memory, 25
- receiving from device, 21
- registering callbacks, 17
- rejecting incoming applications, 23
- remaining memory, 25
- replying to messages, 19
- resetting the handheld, 42
- sending messages, 20
- sending synchronous messages, 20
- setting alarm, 28
- setting configuration settings, 32
- setting process attributes, 21
- setting the date, 28
- setting the language, 43
- setting the time on the handheld, 28
- setting the timer, 29
- stopping applications for a specified period of time, 30
- structures, 7
- switching tasks, 16, 20
- task yielding, 22
- terminating threads, 22
- testing configuration settings, 32
- turning power off, 42, 43
- unrecoverable errors, 35
- waiting for specific message, 23
- System structures, 7

T

- task ID, 14, 15, 16, 39
- task switching, 16, 20, 21, 113
 - preserving the display, 84, 89
- task yielding, 22, 39
- TASK_LIST_CHANGED, 113
- text See displaying
 - text
- threads, 14
 - defined, 13
 - terminating, 22
- THUMB_CLICK, 115
- THUMB_ROLL_DOWN, 115
- THUMB_ROLL_UP, 115
- THUMB_UNCLICK, 115
- TIME structure, 26
- timer
 - cancelling, 27
 - device events, 114
 - ID, 27
 - setting, 29, 114
- tone generator, 30, 31, 32
- trackwheel, 27
- tunes
 - getting name, 31
 - getting number, 31

U

- unrecoverable error, 35

V

- vibrator device, 30, 31, 32

Index



© 2002 Research In Motion Limited
Produced in Canada