

# **BlackBerry Software Development Kit**

**Version 2.5**

**Developer Guide**

BlackBerry Software Development Kit Version 2.5 Developer Guide  
Last modified: 18 July 2002

Part number: PDF-04632-001

At the time of publication, this documentation complies with RIM Wireless Handheld 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](http://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.

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

Published in Canada

# Contents

	<b>About this guide.....</b>	<b>5</b>
	Audience .....	5
	Developer support .....	5
	Other documentation .....	5
<b>CHAPTER 1</b>	<b>Getting started .....</b>	<b>7</b>
	About the BlackBerry SDK .....	8
	Installing the BlackBerry SDK.....	10
	Configuring Microsoft Visual Studio.....	11
<b>CHAPTER 2</b>	<b>Loading applications .....</b>	<b>15</b>
	Checking .dll files .....	16
	Loading applications for testing .....	16
	Deploying applications .....	25
<b>CHAPTER 3</b>	<b>Using the simulator.....</b>	<b>29</b>
	About the simulator .....	30
	Starting the simulator.....	30
	Using the simulator .....	35
	Debugging hints.....	45
<b>CHAPTER 4</b>	<b>Programming overview .....</b>	<b>47</b>
	Understanding the application environment .....	48
	Operating system .....	48
	API hierarchy .....	53
	Application development steps .....	54

<b>CHAPTER 5</b>	<b>Writing an application .....</b>	<b>55</b>
	Basic program structure.....	56
	Defining an entry point.....	56
	Registering the application.....	57
	Entering the message loop.....	58
	Adding the application to the Home screen.....	59
	Setting up a basic program structure.....	60
	Minimizing memory usage .....	63
	Defining version information .....	64
	Other example programs.....	66
<b>CHAPTER 6</b>	<b>Operating system services .....</b>	<b>67</b>
	Inter-process communication.....	68
	File system services .....	70
<b>CHAPTER 7</b>	<b>Radio communications .....</b>	<b>77</b>
	About the Radio API .....	78
	Data packets.....	79
	Transmitting packets.....	79
	Receiving packets.....	80
<b>CHAPTER 8</b>	<b>Writing UI applications .....</b>	<b>81</b>
	Screens.....	82
	Menus .....	83
	Fields.....	84
	Status boxes .....	85
	Dialog boxes .....	86
<b>CHAPTER 9</b>	<b>Messaging.....</b>	<b>87</b>
	About the Messaging API.....	88
	Using the Messaging API.....	89
<b>CHAPTER 10</b>	<b>Ribbon and Options.....</b>	<b>93</b>
	Ribbon .....	94
	Options.....	95

<b>CHAPTER 11</b>	<b>Database tutorial .....</b>	<b>97</b>
	Database API classes .....	98
	Database API features .....	100
	Setting up a database.....	101
	Storing contacts .....	103
	Displaying contacts in a list.....	103
	Defining another contact view .....	105
	Editing a contact.....	106
	Displaying a list using different views .....	107
	Updating a list .....	108
	UI/Database interaction .....	109
	Editing a contact.....	109
	Adding an email address.....	110
	Saving a contact.....	111
	Removing an email field .....	112
<b>CHAPTER 12</b>	<b>Bitmaps, fonts, and sounds .....</b>	<b>115</b>
	Creating bitmaps .....	116
	Converting existing bitmaps .....	117
	Creating custom fonts .....	118
	Resource .dll files .....	120
<b>CHAPTER 13</b>	<b>C library compatibility .....</b>	<b>125</b>
	Summary of C compatibility .....	126
	Compatible functions .....	127
	Incompatible functions .....	127
	<b>Index .....</b>	<b>131</b>



# About this guide

This guide explains how to use the BlackBerry Software Development Kit (SDK) to develop applications for the RIM Wireless Handheld™.

This guide explains these topics:

- installing and configuring the development environment
- testing applications using the simulator
- loading applications onto a handheld
- steps for writing, testing, and debugging applications

## Audience

This guide assumes that you have experience with C++ programming using Microsoft Visual Studio.

## Developer support

For technical updates, FAQs, and a developer's discussion forum, visit the BlackBerry Developer Zone at <http://www.blackberry.net/developers>.

## Other documentation

The BlackBerry SDK also includes the following documentation:

- README.txt

The README.txt file is installed with the BlackBerry SDK. It provides information on any known issues and workarounds, as well as documentation updates.

## About this guide

- *API Reference Guides*

Reference guides are provided for each application programming interface (API). These following guides provide detailed descriptions of functions, structures, and error codes:

- *Address Book API Reference Guide*
- *AutoText API Reference Guide*
- *Operating System API Reference Guide*
- *Database API Reference Guide*
- *Desktop API Reference Guide*
- *Messaging API Reference Guide*
- *Radio API Reference Guide for Mobitex*
- *Radio API Reference Guide for DataTAC*
- *Remote Address Lookup API Reference Guide*
- *Ribbon and Options API Reference Guide*
- *System Utilities API Reference Guide*
- *User Interface API Reference Guide*

# *Chapter 1* **Getting started**

This section provides information on the following topics:

- About the BlackBerry SDK
- Installing the BlackBerry SDK
- Configuring Microsoft Visual Studio

## About the BlackBerry SDK

The RIM Wireless Handheld features a 32-bit Intel 386™ processor that uses the same instruction set as a Windows computer. The handheld has its own multitasking operating system, with built-in messaging.

The BlackBerry SDK provides a set of APIs and tools for you to develop C++ applications to run on the RIM Wireless Handheld, version 2.0 or later.

The BlackBerry SDK package includes the following items:

- API libraries
- simulator
- sample applications

### APIs

The following table lists the API libraries that are included in the BlackBerry SDK. Most applications need to use the Operating System, UI Engine, and Ribbon APIs. Paths to library files are shown relative to the SDK `lib\datatac` and `lib\mobitex` directories.

API/library	Description	Documentation
Address Book API <code>Address.lib</code>	Create, edit, and retrieve contact information from the Address Book database.	<i>Address Book API Reference Guide</i> "address.h"
AutoText API <code>AutoText.lib</code>	Customize the editing of user text, using the AutoText application.	<i>AutoText API Reference Guide</i> "AutoText.h"
File system API <code>..\RIMOS.lib</code>	Access files on the handheld file system.	<i>Operating System API Reference Guide</i> "FileSys.h"
Database API <code>Database.lib</code>	Define new records and create, edit, and retrieve information stored in the database.	<i>Database API Reference Guide</i> "database.h" "DataBuffer.h" "storage.h"
Event Logger API <code>EventLogger.lib</code>	Provide a standardized method for recording events in the handheld's persistent store.	<i>System Utilities API Reference Guide</i> "iEventLogger.h" "iEventViewer.h"
Keypad API <code>..\RIMOS.lib</code>	Customize the keypad. Actual keystrokes are passed as messages.	<i>Operating System API Reference Guide</i> "KeyPad.h"

## About the BlackBerry SDK

API/library	Description	Documentation
LCD API ..\RIMOS.lib	Manages low-level display functions, screen buffers, and string management; use the UI APIs for dialog boxes, menus, and edit classes.	<i>Operating System API Reference Guide</i> "LCD_API.h"
Messaging API message.lib	Send email and other messages.	<i>Messaging API Reference Guide</i> "msg_api.h"
Options API ribbon.lib	Control system-wide programming of features such as the date, time, and screen/keyboard settings.	<i>Ribbon and Options API Reference Guide</i> "Options.h"
Radio API ..\RIMOS.lib	Gain access to radio network and status of packet communications.	<i>Radio API Reference Guide</i> "datatac.h" "mobitex.h"
Registrar API Registrar.lib	Manage registration and instantiation of objects, and object lifetimes and memory allocation.	<i>System Utilities API Reference Guide</i> "ibase.h" "iptr.h" "istr.h"
Remote Address Lookup API	Search for and retrieve entries from a company or third-party Global Address List or other directory.	<i>Remote Address Lookup API Reference Guide</i> "iDirectoryLookupQuery.h" "iDirectoryLookupClient.h" "iDirectoryLookupManager.h"
Ribbon API ribbon.lib	Register an application in the functions list on the Home screen.	<i>Ribbon and Options API Reference Guide</i> "ribbon.h"
Serial Comm API ..\RIMOS.lib	Access and configure the serial port on the handheld.	<i>Operating System API Reference Guide</i> "Comm.h"
System API ..\RIMOS.lib	Manage threads, messaging, and task switching, memory allocation, timers, and the system clock.	<i>Operating System API Reference Guide</i> "Rim.h"
String Utilities API ..\utilities.lib	Use common string utilities not provided in the standard C routines.	<i>System Utilities API Reference Guide</i> "utilities.h"
User Interface API UI32.lib	Manage high-level display of information and user input, such as creating screens, menus, and dialogs.	<i>UI Engine API Reference Guide</i> "UI32.h"

### Tools

The BlackBerry SDK adds an application wizard to Microsoft Visual Studio for RIM application projects.

In addition, the SDK provides a GUI-based simulator for testing your applications. The simulator can be integrated into Microsoft Visual Studio to use its full suite of debugging tools.

In addition, the SDK provides a command-line utility for loading applications onto a handheld.

## Installing the BlackBerry SDK

### System requirements

Your computer must meet the following requirements to support the RIM SDK:

- Operating system — Windows 95/98, Windows ME, Windows NT, Windows 2000, or Windows XP
- Development environment — Microsoft Visual Studio 6.0 or later, or Microsoft Windows Development Environment

### To install the BlackBerry SDK

You must install Microsoft Visual Studio 6.0 before you install the BlackBerry SDK.

1. Double-click the installation icon for the BlackBerry SDK. The InstallShield Wizard Welcome screen appears.
2. Click **Next**. The License Agreement for the BlackBerry SDK appears.
3. Read the license agreement carefully. If you do not agree to its terms, select **I do not accept the terms of this license agreement**; the installation process stops. If you agree to its terms, select **I accept the terms of this license agreement**.
4. Click **Next**. The Customer Information screen appears.
5. Type your user name and company name.
6. From the options at the bottom of the screen, select whether to restrict the use of the SDK to your own user account or to allow any user on this computer to use the SDK.
7. Click **Next**. The Setup Type screen appears.
8. For most installations, leave the **Complete** option selected.
9. Click **Next**.
10. Click **Install**. The InstallShield Wizard installs the SDK.
11. Click **Finish**. The installation process completes.

## Directory structure

The default installation location for the SDK is C:\Program Files\Research In Motion\BlackBerry Handheld SDK 2.5.0.

The BlackBerry SDK has the following directory structure.

Directory	Description
App_samples	Contains subdirectories with sample code that demonstrates how to use some of the RIM Wireless Handheld application APIs.
desktop	Contains header and library files for the Desktop API.
device	Contains operating systems for the handhelds.
dll	Contains application and resource .dll files for the simulator and the handheld; these are subdivided into dataTac and mobiTex directories.
Include	Contains required header files.
lib	Contains libraries for building applications; network-specific libraries are subdivided into dataTac and mobiTex directories.
OS_samples	Contains example programs that demonstrate aspects of the RIM Wireless Handheld operating system. The files ping.c and dt_ping.c are specific to the Mobitex and DataTAC networks, respectively.
Simulator	Contains the RIM Wireless Handheld simulator binaries, .ini file, and legacy files that might be required by some applications.
tools	Contains Windows-based tools provided with the SDK, including programmer.exe.

## Configuring Microsoft Visual Studio

The installation process for the BlackBerry SDK creates a “BlackBerry Application” project type in Microsoft Visual Studio. This project type defines the appropriate settings for the project within the Visual Studio development environment.

You must be familiar with Microsoft Visual Studio to develop applications. You must purchase and become familiar with this application separately from the BlackBerry SDK.

The BlackBerry SDK is designed to make use of the libraries and features in Microsoft Visual Studio, version 5.0 or later.

## Chapter 1: Getting started

- Handheld applications are built as Windows .dll files, but they are not used as Windows applications. The final .dll file is stripped of extraneous information and then ported into the handheld operating system.
- Because Windows sees the handheld applications as .dll files, Visual Studio enables you to use nearly all development facilities that are available for Windows, including the development environment, source level debugging, and breakpoints.

## Project settings

The **BlackBerry Application** project type has the following settings.

**C/C++** tab:

- In the **Preprocessor** category, the `include` and `include\internal` directories are added to **Additional Include Directories** field.
- In the **Code Generation** category, **Struct member alignment** is set to **2 bytes** and **Processor** is set to **80386**.
- In the **C/C++ Language** category, all options are disabled so that Windows-specific code is not invoked.

**Link** tab:

- In the **General** category, the files `OsEntry.obj`, `RimOs.lib`, and `libc.lib` are included in the **Object/library modules** field. The **Ignore all default libraries** option is enabled.
- In the **Input** category, the full path to the SDK `lib` directory is added in the **Additional library** field.

## Creating projects

The BlackBerry SDK installs a project wizard into the development environment.

To create a project using the wizard, complete the following steps:

1. Start Visual C++ 6.0.
2. On the **File** menu, click **New**. The New screen appears.
3. Click the **Projects** tab.
4. Select **BlackBerry Application** and type a project name.
5. Select a network (**DataTAC** or **Mobitex**).
6. Select the persistence of the application. The following table explains the options.

Option	Meaning
Persist	The program never really exits, and resources are not freed when the user returns to the ribbon. The generated code uses <code>RibbonRegisterApplication()</code> to register with the ribbon; program identifiers are set up in <code>PagerMain()</code> .
Exit (dynamically loaded/unloaded)	The program exits and resources are freed when the user returns to the ribbon. The generated code sets up program identifiers in <code>PagerFunc()</code> , and registers that function using <code>RibbonRegisterFunction()</code> .

7. Click **Finish**.
8. Click **OK** to confirm the options.  
The new project appears in the Microsoft Visual Studio project workspace.
9. In the Microsoft Visual Studio If you go to the File View, double-click the source file for the project.

Note that the App Wizard has created a template for a typical handheld application.

The Application Wizard creates code that performs the following tasks:

- defines `PagerMain()` and, for non-persistent applications, `PagerFunc()`
- instantiates a **UIEngine** object (`g_UIEngine`)
- instantiates a Screen object (`MyScreen`) with these member functions:  
`MyScreen::OnKey()`  
`MyScreen::OnMenuItem()`  
`MyScreen::OnMessage()`
- declares these global fields:  
`AppStackSize`  
`g_menuItemCount`  
`g_MenuStrings` and an enumerated type for menus  
`VersionPtr`

If your application will not use the UI Engine API to display a user interface, you should rewrite this code.

## Setting project properties

When you create a new project for a handheld application, you must set the debugging options to run applications in the simulator.

1. In Visual C++ 6.0, select a project.
2. On the **Project** menu, click **Settings**.  
A Project Settings screen appears.
3. From the **Settings For** drop-down list, select **All Configurations**.
4. Click the **Debug** tab.
5. In the **Executable for debug session** field, type the full path to the OSLoader.exe. For example:

```
C:\Program Files\Research In Motion\ BlackBerry Handheld SDK 2.5\
simulator\OSLoader.exe
```

6. In the **Program arguments** field, type one of the following file names:

- **OsPgrMb.dll** (RIM Wireless Handheld 950™)
- **OsHHMb.dll** (RIM Wireless Handheld 957™)
- **OsPgrDt.dll** (RIM Wireless Handheld 850™)
- **OsHHDt.dll** (RIM Wireless Handheld 857™)

You can also type other command line options (refer to “Using the simulator” on page 35).

7. In the **Working directory** field, specify the folder in which the simulator .dll files are located.
8. Click **OK**.

When you debug this project in Visual C++ 6.0, it runs the application in the simulator.

Refer to “Using the simulator” on page 35 for more information on using the simulator.

# *Chapter 2*

# Loading applications

This section provides information on the following topics:

- Checking .dll files
- Loading applications for testing
- Deploying applications

## Checking .dll files

Before you load .dll files onto the handheld, you can use the DLL utility to view statistics, such as the amount of flash memory and RAM that the .dll file will require on the handheld.

The DLL utility is `DllUtil.exe` in the SDK `tools` folder.

There are two forms of the command:

```
DllUtil SIZE [-R] files [files*]  
DllUtil VER files [files*]
```

The *files* argument is a list of DLL or OS files, and can contain wildcards (\* and ?).

### SIZE command

With the `SIZE` command, `dllutil` displays the flash memory, RAM, and thunk resources required by the DLL on the handheld.

`-R`

When this switch is used, `dllutil` displays the size of the .dll files without the DLL relocation information.

### VER command

With the `VER` command, `dllutil` displays version information. Beside each valid file, it lists versions for both imported and exported APIs. It also lists valid operating systems along with file version information.

## Loading applications for testing

The `Programmer.exe` tool provides a command line tool to add or update compiled Windows .dll files on a handheld.



Note: This tool is intended for use by developers for development and testing purposes only. To deploy production applications, you should use the Application Loader, which is part of the BlackBerry Desktop Manager. Refer to "Deploying applications" on page 25 for more information.

1. Insert the handheld into the cradle.
2. Exit the BlackBerry Desktop Manager.

The `programmer.exe` utility cannot connect to the handheld if the Desktop Manager utility is already running. Only one program can connect with the handheld at a time.

3. Move to the `tools` folder in the SDK installation folder.

4. Type:

**PROGRAMMER [-Pport] [-Sspeed] [-Wpassword] <command>**

where:

<port> is the serial port to which the handheld is connected (default is COM1).

<speed> is the bit rate speed to the serial port (default is 115200).

<password> specifies the password for your handheld, if one has been set.

<command> is one of the following command options.

## Command options

The following command options are supported:

Command	Description
alloc [-e] [-d sectors] [-a sectors]	moves breakpoint between application memory and file system memory
batch filename	runs programmer . exe commands stored in a file
dir [-s]	lists applications currently on the handheld
erase [-A -E] apps	erases applications, OS, or both from the handheld
help [command] help error	displays command usage information
load [-s] [-G] [-V] [-M] [-d] apps or ( group )	loads applications or groups of applications on the handheld
map [-f] [-r]	displays detailed flash memory and RAM maps
ver	lists applications that are currently on the handheld, including version information
wipe [-F   -A]	irreversibly erases applications, the file system, or both from handheld

## ALLOC command

Moves breakpoint between application memory and file system memory.

```
PROGRAMMER ALLOC [-E] [-D <sectors>] [-A <sectors>]
```

<b>Parameters</b>	-E	Specifies the File Allocation Sector to be erased before writing an entry.
	-D sectors	Specifies the new size of the file (or data) area in flash memory sectors; the minimum size is 1.
	-Asectors	Specifies the new size of the OS and applications area in flash memory.

**Description** This command writes a new entry to the Flash Allocation Log and moves the breakpoint between the file and application areas of memory. It is possible to move only one, but then you would have an unused sector.

The command only reduces the size of an area if the sectors to be removed are empty. This is rarely a problem with the OS and applications area, but might be a problem with the File area, because there are no guarantees about the placement of information.

When the File Area size is decreased, at least one blank sector must remain in the area. You are advised to back up and pack your data before issuing the ALLOC command.

## BATCH command

Run a batch file containing programmer commands.

```
PROGRAMMER BATCH batchfile
```

<b>Parameters</b>	<i>batchfile</i>	Name of a file containing commands.
-------------------	------------------	-------------------------------------

**Description** The batch command enables multiple commands to be placed in a file and run with a single, short command. This command is useful if you need to perform the same process repeatedly.

The batch file can contain one or more of the `load`, `erase`, `dir`, or `batch` commands. The results are committed to the handheld only if all commands are successfully completed.

The maximum length of a line is 256 characters. Single commands can be broken into multiple commands, if you have long file names.

## DIR command

List applications that are currently on the handheld.

```
PROGRAMMER DIR [-S]
```

**Parameters**    -S        Specifies a short listing of only the application names.

**Description**    The `dir` command generates a listing of the applications currently loaded on a handheld. Unless the `-S` option is specified, this listing includes the names of the applications and the amount of flash memory and RAM occupied by the applications. The applications are grouped as they are grouped on the handheld.

**Example**        The following command lists the applications on the handheld:

```
PROGRAMMER DIR
```

## ERASE command

Erases applications, or OS, or both from the handheld.

```
PROGRAMMR ERASE -A
PROGRAMMR ERASE app_names
```

**Parameters**    -A                Specifies that all applications and the application environment are to be deleted.

                  app\_ names    Names of specific applications to be deleted

**Description**    The Erase command erases the applications that are currently loaded on a handheld. The names are not case sensitive and can be obtained using the `PROGRAMMER DIR` command.

The space occupied by erased applications is only reclaimed when the entire group in which it resides is erased.

**Example**        The following command erases all applications on the handheld:

```
PROGRAMMER ERASE -A
```

The following command erases only the Address Book application:

```
PROGRAMMER ERASE ADDRESS.DLL
```

## HELP command

```
PROGRAMMER HELP [command]  
PROGRAMMER HELP errors
```

**Parameters**

command	Name of the command for which you want help.
errors	Causes programmer.exe to list descriptions of several error messages that are not self-explanatory

**Description** The Help command invokes the built-in Help system. Help for a specific command can be obtained by specifying it as an option. Help about error messages can be obtained by specifying the errors command.

If the output is not redirected to a file, the help system uses a built-in paging system. Press any key at the <MORE> prompts.

**Example** For help on the Load command, type:

```
PROGRAMMER HELP LOAD
```

For help on errors, type:

```
PROGRAMMER HELP ERRORS
```

For the main help page, type:

```
PROGRAMMER HELP HELP
```

## LOAD command

Loads new applications or application environment onto the handheld.

```
PROGRAMMER LOAD [-S] [-G] [-V] [-M] [-D] apps or groups
```

**Parameters**

-S	Specifies that symbol information for all new applications should be appended to a debug.dat file in the current directory.
-G	Specifies that the first application or group of applications should be grouped with the last group found on the handheld.
-V	Specifies checking file versions. By default, programmer checks API versions and dependencies only.

-M	Specifies that mappings of unresolved OS calls should be displayed. The Application Loader maps unresolved OS calls to an internal API, <code>RimCatastrophicAPIFailure()</code> . In all cases, other unresolved links fail; only unresolved OS calls can be mapped.
-D	Specifies that unresolved OS calls should not be mapped to <code>RimCatastrophicAPIFailure()</code> .
apps or groups	One or more files or groups of files to be loaded onto the handheld. Individual files are specified alone. Groups of files are enclosed in parentheses, brackets, or braces. There must be spaces surrounding the brackets, as in the example.

**Description** The Load command loads new applications or the application environment onto the handheld. Any old applications with the same name are erased. If an application is to be replaced separately from other applications on the handheld, place it in its own group.



**Note:** When loading or replacing the application environment (`pagerx50.bin` or `pagerx57.bin`), it must be specified first on any load command.

### Grouping programs

Because the flash memory can only be erased one 64-KB sector at a time, any application that is to be erased and have its space reclaimed must not overlap with other applications in the same 64-KB sector. Grouped applications are contiguous, without regard to the 64-KB sector boundaries. As such, when invalidating an application that is part of a group, the space cannot be reclaimed without erasing other applications as well. Applications that are not grouped each occupy one or more 64-KB sectors, with the remainder of the last sector used being wasted.



**Note:** The ability to group programs is specific to `programmer.exe`. Although the desktop loader program will recognize groups already on a RIM Wireless Handheld, it cannot load files in groups.

**Example** The following command loads the application environment and applications onto the handheld:

```
PROGRAMMER LOAD PAGER950.BIN UI32.d11 ( Address.d11 AutoText.d11  
Message.d11 Options.d11 Transport_RTP.d11 )
```

The following command loads a new calculator application, grouping it with the other applications:

```
PROGRAMMER LOAD -G Calculator.d11
```

## MAP command

Displays detailed maps of flash memory and RAM.

PROGRAMMER MAP [-F] [-R]

- Parameters**
- F Specifies the flash memory map.
  - R Specifies the RAM map.

**Description** If no options are given or both options are given, Map displays both maps. For each item in memory, the map displays its beginning address, ending address, total size and name. Numeric values are given in hexadecimal.

The output looks like this:

```
RIM Wireless Device Command-Line Programmer Version 1.0.0.21
Copyright 2002 Research In Motion Limited
Connecting to device...
Connected
Reading Configuration      [##### 100 #####]
```

```
Release -4
Hardware: "RIM Handheld for Mobitex (Intel)"
```

FLASH MAP (available flash = 1984 KB)

from	to	size	name
3fb0000	3feffff	40000	PAGER957.EXE
3fa1a90	3faffff	e570	Address.dll
3fa1250	3fa1a8f	840	AddressAttachment.dll
3f9b3e0	3fa124f	5e70	AutoText.dll
3f9a880	3f9b3df	b60	CoreApi.dll
3f90bf0	3f9a87f	9c90	CryptoBlock.dll
3f873c0	3f90bef	9830	Database.dll
3f5c570	3f873bf	2ae50	Message.dll
3f45700	3f5c56f	16e70	ribbon.dll
3f30260	3f456ff	154a0	SecureTransport.dll
3f2e6d0	3f3025f	1b90	SerialDbAccess.dll
3f29350	3f2e6cf	5380	Transport_MDP.dll
3f0c310	3f2934f	1d040	UI32.dll
3f06a20	3f0c30f	58f0	Calculator.dll
3ee7390	3f06a1f	1f690	Calendar.dll
3ee3c10	3ee738f	3780	MemoPad.dll
3ee0c20	3ee3c0f	2ff0	RimTris.dll
3eddf30	3ee0c1f	2cf0	Solitaire.dll
3ed0000	3eddf2f	df30	[free]
3e00000	3ecffff	d0000	[free]

RAM MAP (available RAM = 560 KB)

from	to	size	name
580000	581f23	1f24	Address.dll
581f24	581f5b	38	AddressAttachment.dl
581f5c	58292b	9d0	AutoText.dll
58292c	582a8e	163	CoreApi.dll
582a8f	58425a	17cc	CryptoBlock.dll
58425b	584522	2c8	Database.dll
584523	588d32	4810	Message.dll
588d33	58c0de	33ac	ribbon.dll
58c0df	58cec2	de4	SecureTransport.dll
58cec3	58cfc6	104	SerialDbAccess.dll
58cfc7	58eb66	1ba0	Transport_MDP.dll
58eb67	58f62e	ac8	UI32.dll
58f62f	58fcfa	6cc	Calculator.dll
58fcfb	590e0e	1114	Calendar.dll
590e0f	591096	288	MemoPad.dll
591097	5918a6	810	RimTris.dll
5918a7	59196e	c8	Solitaire.dll
59196f	5f0fff	5f691	[free]
5f1000	60bfff	1b000	PAGER957.EXE

Disconnected

**Example** The following command shows both maps:

```
PROGRAMMER MAP
```

## VER command

```
PROGRAMMER VER
```

**Description** The ver command lists applications that are currently loaded on a handheld, including version information. Version information includes release tags and build times. The applications are grouped as they are grouped on the handheld.

**Example** The following command lists the applications on the handheld, including release tags and build times:

```
PROGRAMMER VER
```

## WIPE command

Wipes the file system memory or application memory or both.

```
PROGRAMMER WIPE [-F | -A]
```

## Chapter 2: Loading applications

- Parameters**
- F Specifies that the file system should be wiped.
  - A Specifies that the application area should be wiped.

**Description** If no option is specified, both the file system and application are wiped.

## Troubleshooting

Most error messages are self-explanatory. The following errors require more discussion.

### Error: Unable to connect to device

An error occurred trying to initiate communications with the handheld. Make sure that it is connected to the computer properly.

### Error: Insufficient flash or RAM

There is not enough flash memory or RAM remaining to load a new application. Make sure that you have erased any old applications. If you have been erasing and loading often, you might have fragmented your flash memory space. In this case, erase all applications and start again. Loading applications as part of the same group causes them to occupy flash memory more efficiently.

### Error: Relocation failed

It was not possible to relocate an application. This typically occurs if the application environment (`pager950.bin`) is not specified first in a LOAD command. When loading or replacing the application environment, the application environment must always be specified first in the LOAD command.

### Error: Bad load

This error occurs when a LOAD is interrupted or performed improperly. This is a serious error that requires the handheld to be reset and its operating system to be reloaded. To reset the handheld, insert a small pointed instrument, such as a paper clip, into the reset hole on the back of the handheld. To reload the operating system, use the LOAD command. Refer to "LOAD command" on page 20 for more information.

### Error: Not all imports resolved

An application is requesting imports from another application that cannot be exported by the other application. Verify that you are loading applications that provide the exports that you need.

## Deploying applications

The Application Loader, which is part of the BlackBerry Desktop Manager, enables users to load new applications onto the handheld.

To deploy applications in this manner, you must create an application loader (.alx) file for each application and then distribute the .alx and .dll files to your users.

For information on using the Application Loader, refer to the *Application Loader Online Help*.

### Sample .alx files

In a text editor, such as Microsoft Notepad, create a new .alx file for your application. The .alx file uses extensible markup language (XML) format. This is an example of an .alx file for one application:

```
<loader version="1.0">
  <application id="com.rim.samples.device.contacts">
    <name>Sample Contacts Application</name>
    <description>Provides ability to store a list of contacts.
    </description>
    <version>1.0</version>
    <vendor>Research In Motion</vendor>
    <copyright>Copyright 1998-2002 Research In Motion</copyright>
    <language langid="0x000c">
      <name>Application D'Échantillon Pour des Contacts</name>
      <description>Enregistre une liste de contacts.</description>
    </language>
    <fileset Java="1.0">
      <directory>samples/contacts</directory>
      <files>
        net_rim_contacts.dll
        net_rim_resource.dll
        net_rim_resource__en.dll
        net_rim_resource__fr.dll
      </files>
    </fileset>
  </application>
</loader>
```

This is an example of a .alx file for an application with nested modules:

```
<loader version="1.0">
  <application id="net.rim.sample.contacts">
    <name>Sample Contacts Application</name>
    <description>Provides the ability to store a list of contacts.
  </description>
    <version>1.0</version>
    <vendor>Research In Motion</vendor>
    <copyright>Copyright 1998-2001 Research In Motion</copyright>
```

## Chapter 2: Loading applications

```
<language langid="0x000c">
  <name>Application D'Échantillon Pour des Contacts</name>
  <description>Enregistre une liste de contacts.</description>
</language>
<fileset Java="1.0">
  <directory>samples/contacts</directory>
  <files>
    net_rim_contacts.cod
    net_rim_resource.cod
    net_rim_resource__en.cod
    net_rim_resource__fr.cod
  </files>
</fileset>
<application id="net.rim.sample.contacts.mail">
  <name>Sample Module for Contacts E-Mail Integration</name>
  <description>Provides the ability to access the email
    applicaton</description>
  <version>1.0</version>
  <vendor>Research In Motion</vendor>
  <copyright>Copyright 1998-2001 Research In Motion</copyright>
  <language langid="0x000c">
    <name>Application D'Échantillon</name>
    <description>Utiliser l'application de E-mail</description>
  </language>
  <fileset Java="1.0">
    <directory>samples/contacts</directory>
    <files>
      net_rim_contacts_mail.cod
    </files>
  </fileset>
</application>
</application>
</loader>
```

## Format of .alx files

The following table describes each element (tag) and attribute. All elements are mandatory unless otherwise noted.

Element	Attributes	Description
loader	version	The loader element contains one or more application elements. The version attribute specifies the version of the Application Loader. The version in this release is 1.0.
application	id	<p>The application element contains the elements for a single application.</p> <p>The application element can also contain additional nested application elements. Nesting enables you to load all prerequisite modules when an application is loaded.</p> <p>The id attribute specifies a unique identifier for the application. You should use an ID that includes your company domain, in reverse, for uniqueness (for example, com.rim.samples.device.contacts).</p>
name	—	The name element provides a descriptive name for the application, which appears in the Application Loader. It does not appear on the handheld.
description	—	The description element provides a brief description of the application, which appears in the Application Loader. It does not appear on the handheld.
version	—	The version element provides the version number of the application. This version number appears in the Application Loader.
vendor	—	The vendor element provides the name of the company that created the application. The vendor name appears in the Application Loader.
copyright	—	The copyright element provides copyright information, which appears in the Application Loader.

## Chapter 2: Loading applications

Element	Attributes	Description
language	langid	<p>The language tag enables you to specify additional languages for application information that appears in the Application Loader when the desktop language is other than the default language.</p> <p>You can nest the name, description, version, vendor, and copyright tags in the language tag. You can specify multiple language tags, one for each language that you want to support.</p> <p>The langid attribute specifies the Win32 langid code for the language to which this information applies. For example, some Win32 langid codes are: 0x0009 (English), 0x0007 (German), 0x000a (Spanish), 0x000c (French).</p>
fileset	model	<p>The fileset element includes one directory element and one files element. It specifies a set of .dll files, in a single directory, to load onto the handheld. If you need to load files from more than one directory, you can include one or more fileset elements in the .alx file.</p> <p>The model attribute enables you to load different applications or modules depending on the type of handheld. The model attribute is optional.</p>
directory	—	<p>The directory element provides the location of a set of files. The directory element is optional. If you do not specify a directory, the files must be in the same location as the .alx file.</p> <p>You can specify a directory element for an application or for each fileset. The directory is specified relative to the location of the .alx file.</p>
files	—	<p>The files element provides a list of one or more .dll files, in a single directory, to load onto the handheld for an application.</p>

# *Chapter 3* Using the simulator

This section provides information on the following topics:

- About the simulator
- Starting the simulator
- Using the simulator
- Debugging hints

## About the simulator

The simulator included with the SDK enables you to test applications without having to load the .dll files onto an actual handheld.

The simulator supports all of the functionality of the radio modem. You can connect the simulator to a Radio Access Protocol (RAP) modem through the computer serial port to enable simulated applications to communicate over the live DataTAC network.

## Starting the simulator

You can start the simulator in one of three ways:

- integrate the simulator with Microsoft Visual Studio
- use the Windows shortcut
- use the command line

Each application is built into a separate .dll file. You load applications into the simulator by selecting one or more .dll files.

There are two RIM Wireless Handheld simulator programs: `Simulator.exe` and `OsLoader.exe`.

- `Simulator.exe` provides a Windows interface and provides consistency between sessions, because configuration information is stored in the `simulator.ini` file.
- `OsLoader.exe` provides a command-line interface and is normally used as the debug executable in the Microsoft Visual Studio IDE.

For each program, you must specify the platform that you want to simulate and the applications that you want to load.

- Platform .dll files are in the `simulator` directory
- Application .dll files are in the `d11/datatac` and `d11/mobitex` directories

There are differences between each network simulation, based on the differences between the networks themselves. Refer to the *Radio API Reference Guide* for information such as:

- simulation file protocol
- format of the files that simulate data packets
- limitations of the simulation
- specific behaviors of the simulation
- guidelines for writing a host-side application to simulate the network

## Running the simulator with Microsoft Visual Studio

You can set up the project properties for projects in Microsoft Visual C++ 6.0 so that you can run an application in the RIM Simulator environment during debugging. Refer to "Setting project properties" on page 14 for more information.

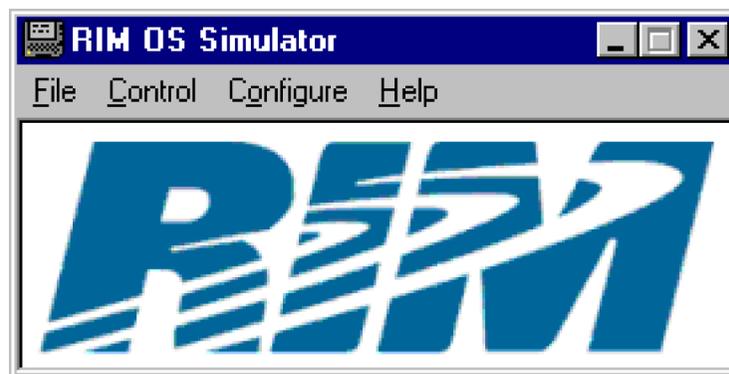
When you debug a project in Visual C++ 6.0, the simulator automatically starts. You can then run the application in the simulator while using all of the debugging facilities, such as breakpoints, in Visual C++ 6.0.

## Running the simulator in Windows

1. On the **Start** menu, select **Programs > BlackBerry Handheld SDK 2.5 > BlackBerry Simulator**.

Alternatively, in the SDK simulator folder, double-click **Simulator.exe**.

The RIM OS Simulator window appears:



RIM OS Simulator window

2. On the **Configure** menu, select the platform that you want to simulate.

If no platforms are listed, click **Platforms**. In the window that appears, click **Add**. In the simulator folder select the appropriate OS .dll file, such as OsPgrDt.dll.



**Note:** When you change platforms, remove the memory dump file (extension .dmp) in the simulator directory. To remove this file, on the **Configure** menu, click **Options**. Click the **Flash** tab, and then click **Erase flash (once)**.

3. On the **Control** menu, clear the **Prompt for options** and **Prompt for applications** options.

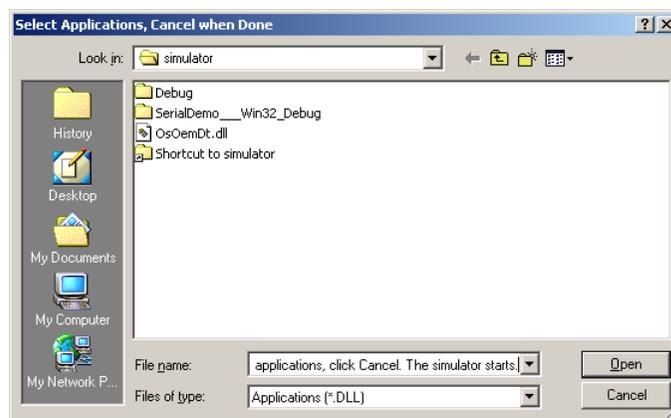
Select these options if you want to set simulator options and select the applications to load each time you run the simulator.

## Chapter 3: Using the simulator

4. On the **Configure** menu, click **Options**. The **Simulation Options** window appears.  
Refer to the *Simulator Online Help* for information on options.
5. Click the **Applications** tab. Beside the **Always load these applications** field, click **Browse**.
6. In the **Select Applications** window, select the application .dll files to load and click **Open**. You can select several applications.
7. After you select applications, click **Cancel**.
8. On the **Control** menu, click **Start Simulation**.  
Refer to “Using the simulator” on page 35 for more information.

## Starting the simulator from a command prompt

1. Set the PATH environment variable to include the folder for all your application .dll files. This helps prevent errors due to dependencies between applications.
2. Using a command prompt, go to the SDK `tools` folder and type:  
**OSLOADER.EXE [Options] OSXXXXX.dll [DLLs]**  
The first parameter must be an OS .dll file, such as `OsPgrMb.dll`. For example:  
**OSLOADER.EXE OsPgrMb.dll SampleApp.dll**  
`OsPgrMb.dll` is the operating system to use and `SampleApp.dll` is the application to load into the simulator. Application .dll files must be included in the PATH environment variable.  
Refer to “Simulator options” on page 33 for more information on options.
3. If you did not specify any .dll files at the command prompt, in the Select Application dialog box, select each .dll file that you want to load and click **Open**.  
After you select the applications you want to load, click **Cancel**.



**Select Applications window**

The simulator starts.

Refer to “Using the simulator” on page 35 for more information.

## Simulator options

The following table summarizes the options for controlling the simulator. Equivalent command line switches and arguments are also given; all options have an equivalent switch.

To set the following options, on the **Configure** menu, click **Options**. The following table organizes options according to the tab in the options dialog box.

Tab	Option	Switch	Description
General	PIN	/RSIM=addr	This option specifies <i>addr</i> as the simulator's address.
	Turn off audio	/B	This option turns off audio.
Flash	Total size	/Fn	This option simulates <i>n</i> KB of flash memory.
	OS and app sectors	/An	This option specifies <i>n</i> 64-KB sectors of flash memory for OS and application storage. By default, the previous amount is preserved unless /E (Erase Flash) is simultaneously specified. Then a default amount is used. This can simulate the result of a PROGRAMMER ALLOC command.
	Filesystem sectors	/Dn	Specifies <i>n</i> 64-KB sectors of flash memory for file system data storage; by default, the previous amount of flash memory is preserved, unless /E (Erase Flash) is specified.
	Erase flash (once)	/E	Erases flash memory allocation log.
Ports	RAP port	/Rn	Specifies the serial port <i>n</i> for RAP modem.
	Serial port	/Sn	Specifies Windows serial port <i>n</i> in place of handheld's physical serial port; if no port is specified, applications cannot open ports.
Applications	Default application directory		Specifies the directory from which to load applications; on the command line, this is the current working directory when the simulator is invoked.
	Always load these applications		On the command line, any argument that does not begin with / is taken as the file name of an application to load.

### Chapter 3: Using the simulator

Advanced		/C	Disables checking of application .dll files; by default, the loader checks that the .dll files are valid and have no external dependencies, and that each .dll file is loaded only once
		/Mn	Sets memory debugging level, where <i>n</i> is one of: 0—Use only main heap 1—Use private heap and main Windows heap 2—Use both heaps, with bounds checking 3—Use both heaps, with free() pointer checking 4—Use both heaps, free() pointer checking and bounds checking The default is 4 (highest level).
		/N	Deletes the flash memory file system before initialization, instead of resuming from last run (the default).
		/RDIR=d i r	Stores simulated packets in directory d i r. The default is the current directory; two simulations must be pointing at the same directory to exchange packets.
		/T[flags]	Set or suppresses traps. Valid flags are: + (turn on subsequent options) - (turn off subsequent options) W (trap when the application writes to flash) Y (trap when the application yields control) Refer to "Simulating pointer trapping behavior" on page 41 for more information.

To set the following options, click the **Control** menu.

Option	Switch	Description
Prompt for applications	/Pf	Force prompt for more applications.
	/Ps	Always suppress prompt for more applications.

### Loading applications

Each .dll file is a different application. The .dll files you load depend on which handheld applications you want to simulate.

For example, a simple application that uses only the functionality of the base API functions do not depend on other .dll files and can be loaded by itself; however, applications that are built with the UI Engine API require that the ui32.dll file is also loaded.

To simulate an application running with the full handheld messaging application, load all of the .dll files in the directory. You should load the Ribbon and the UI Engine. You can choose to leave out some applications.

Refer to the specific *API Reference Guides* for more information on each of these .dll files.



**Note:** The sdkradio.dll is not a standard application .dll file for the simulator. It is used when you want to use your simulator as a Mobitex modem. (Refer to "Using a physical modem" on page 43 for more information.) Normally you should not load SDKRadio.dll into the simulator.

### Stopping the simulation

To stop the simulation, on the **Control** menu, click **Stop simulation**.

### Exiting the simulator

To exit the simulation, on the **File** menu, click **Exit**.

## Using the simulator

When you run the simulator, a window appears that displays a picture of the handheld. This main window contains pictures of a keypad, a trackwheel, and an LCD. You can operate the keypad and trackwheel from Windows, and the LCD shows you what would be displayed on the actual handheld.

For information on using the handheld itself, refer to the *Handheld User's Guide*. This section explains the differences between the simulator and the actual handheld.



RIM Handheld Simulator for Mobitex

### Using the keyboard

Press the keys on your computer keyboard or click the keys on the simulator keyboard to simulate pressing handheld keys. You do not need to press the ALT key to type numbers and symbols; use the number keys on your computer keyboard.

You can change this input method. On the simulator **Keyboard** menu, click **Pager**. In Pager mode, you can only use the keys on your keyboard that exist on the handheld. To type numbers and symbols, press **CTRL** on your computer keyboard and its associated letter on the handheld.

### Using the ESCAPE key

Handheld users press **ESC** to close the current menu or screen and return to the previous screen.

Press **ESC** on your computer keyboard to simulate pressing **ESC** on the handheld.

### Using the trackwheel

The trackwheel is the handheld user's primary input mechanism. The trackwheel enables users to navigate, view, and select items on each screen. Users roll the trackwheel to scroll through menu items or text, and click the trackwheel to select applications or menu options.

When the simulator window is active, you can roll the wheel button on your mouse to simulate rolling the trackwheel, and click the wheel button to simulate clicking the trackwheel.

If your mouse does not have a wheel button, use the **UP ARROW** and **DOWN ARROW** keys on your keyboard to simulate rolling the trackwheel, and press the **LEFT ARROW** or **RIGHT ARROW** key to simulate clicking the trackwheel.

## Enabling backlighting

On the pager-sized handheld, press **ALT** three times successively to turn on the backlighting; the background of the LCD turns blue.

On the palm-sized handheld, press the silver key.

The backlighting will stay on until an idle period of about ten seconds is detected.

## Setting LCD size

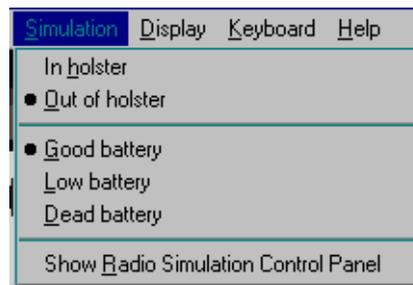
The LCD size can be toggled between full size (with correct aspect ratios), and actual size. To change the size of the LCD on your simulator, on the **Display** menu, click **Change size**.



**Note:** Use actual size to determine how readable a display screen will be on the actual handheld.

## Simulating battery conditions

The radio modem simulator enables you to simulate various battery conditions of the radio modem, for applications in which the RIM 802D Radio Modem is battery-powered. On the **Simulation** menu, select a battery condition: **Good battery**, **Low battery**, **No battery**.



Simulation menu

## Simulating radio conditions

To display the Radio Simulation control panel, on the **Simulation** menu, click **Show Radio Simulation Control Panel**. The panel is automatically displayed if the `/RSIM` or `/RDIR` command line options are used.

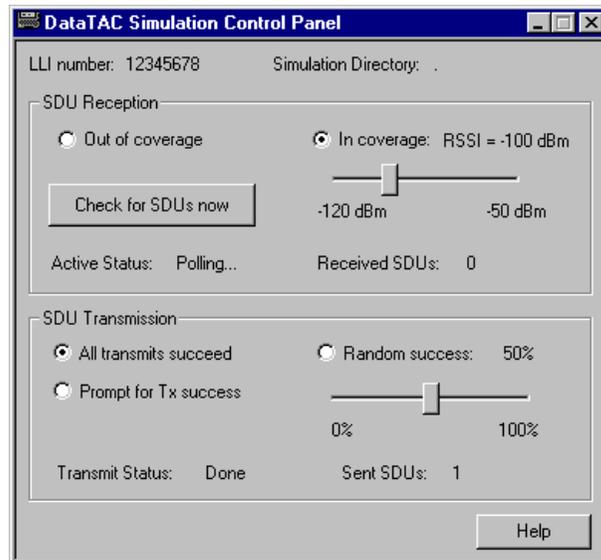
Using the control panel, you can simulate various coverage conditions. This control panel is only useful when you are using the file system to simulate the radio network.

Note the following differences depending on the network being simulated:

- The DataTAC simulation refers to data packets as SDUs, while the Mobitex simulation refers to data packets as MPAKs.

## Chapter 3: Using the simulator

- The DataTAC simulation refers to the handheld identification number as an LLI, while the Mobitex simulation refers to the handheld identification number as the MAN.



**Radio Simulation Control Panel**

The controls are in two groups: simulation of packet reception and simulation of packet transmission.

In these descriptions, *packet* stands for the network-specific packet name.

### MAN/LLI number

The **MAN number** or **LLI number** field indicates which MAN or LLI number the modem is currently using. Only one simulation can exist for each identifier. At startup, the simulator verifies that no other simulation is running with that MAN or LLI number.

### Simulation Directory

The **Simulation Directory** field indicates which directory is used to simulate the network. All packets are communicated across the network using this directory. Two simulators can communicate with each other if they point to the same directory.

### MPAK/SDU reception

The **MPAK Reception** or **SDU Reception** section enables you to simulate network coverage conditions and shows information on received packets.

When you select **Out of coverage**, the modem is out of network contact, and cannot send or receive SDUs.

When you select **In coverage**, you can use the slide bar to control the received signal strength indicator (RSSI) that is reported to applications.

The signal strength varies from -120 dBm to -50 dBm (dBm is decibels with respect to milliwatts). A signal of -120 dBm is extremely weak, and the modem typically loses network coverage before reaching this number. A signal of -50 dBm is stronger than is likely on the actual network.

## Check for MPAKs/SDUs now

Click the **Check for SDUs now** or **Check for MPAKs now** button for the simulator to look for packets immediately. This process is useful in the following situations.

- to increase the frequency with which the simulator checks for packets (by default, the simulator checks the hard drive for packets once every 10 seconds to simulate delays in sending unsolicited packets to a particular modem)
- to increase the speed of simulated retries, so that packets that are sent to a modem that is out of network coverage are returned as undeliverable sooner

## Active status

The **Active Status** field displays one of the following states:

State	Description
Radio Off	The radio is turned off.
Turning Off	The radio is in the process of turning off.
Stop Reception	Radio reception is stopped ( <code>RadioStopReception</code> has been called).
Active	The modem is in network coverage on the network.
Checking...	The radio is checking for SDUs at 10-second intervals, or <b>Check for packets now</b> was clicked.
Checking for 10 sec...	The radio is checking continuously because a packet was recently sent or received.
Out of coverage	The modem is out of network coverage.

## Received MPAKs/SDUs

The **Received MPAKs** or **Received SDUs** field displays the number of packets that have been received from the simulated network.

## MPAK/SDU transmission

The **MPAK Transmission** or **SDU Transmission** section enables you to control the simulated transmission of packets and shows information on sent packets.

### All transmits succeed

When you select this option, every packet submitted for transmission is transmitted successfully to the network. On an actual network, under good network coverage conditions, almost all packets are sent successfully.

### Prompt for Tx success

When you select this option, each time that a packet submitted for transmission, a dialog box appears prompting the user to select whether or not the packet will be transmitted successfully. The packet is pending in the radio until the user selects **Yes** or **No**.

This mode is useful for simulating very long transmission delays. On an actual network, a packet can be pending in the modem for tens of seconds.

### Random success

Using the slide bar, you can select the probability of a particular packet being sent successfully to the network. The delay for successful SDUs is about 1.5 seconds, while the delay for unsuccessful packets is 3 seconds; unsuccessful transmission attempts typically take longer than successful ones.

### Transmit status

The **Transmit Status** field indicates the status of the last or current packet that was submitted for transmission. This packet can be in one of the following states:

State	Description
SDU Pending	The modem is currently attempting to send a submitted SDU.
Transmit Done	The last SDU was sent to the network successfully.
Transmit Failed	The last SDU failed to reach the network.

## Simulating serial I/O

The simulator can use one or two computer serial ports to simulate the handheld serial ports. To enable this option from the command prompt, specify the `/S` option. To enable this option with the Windows simulator, on the **Configure** menu, click **Options** and then click the **Ports** tab.

Refer to "Simulator options" on page 33 for more information.

If this option is not enabled, or the serial port is not available at startup, you cannot open or use the serial ports from applications that are running on the simulator.

If you want to simulate that the handheld is connected to a computer, you must connect the computer serial port to another serial port using a null modem cable.

If you are using the simulator to test connectivity with another RIM handheld application that is running on the same computer, you must configure the software to use different physical serial ports and link the serial ports using a null modem cable.

## Simulating pointer trapping behavior

The log file system and memory model used by the RIM Wireless Handheld means that pointers into flash memory might become invalid after an application writes to the file system or yields to another process. Refer to "Memory-mapped file access" on page 72 for more information.

The simulator simulates this behavior by occasionally moving the simulated file system; the simulator then maps the old location as invalid memory. (The pointer is "stale.") References to used file system pointers that should have been reloaded will result in page faults. (This behavior is useful in debugging.)

The stale pointer trapping behavior is controlled by the /T option. You can specify this option on the command line, or add it to the simulator options. On the **Configure** menu, click **Options**. Click the **Advanced** tab. In the **Extra options** fields, add the /T option.

The /T option can be followed by these symbols:

Flag	Meaning
+	Turn subsequent options on
-	Turn subsequent options off
W	Trap on the application writes to flash memory
Y	Trap when the application yields control

By default, both trap options are turned on.

For example, /TW-Y enables a trap on flash memory write and disables the trap on yield.

### Flash memory simulation files

To better simulate flash memory, the RIM Wireless Handheld simulator loads the flash memory contents from a file on startup and saves them to the file on exit. The state of the simulated flash memory is preserved in a file in the working directory named `0sXxxYy.dmp`, where `Xxx` is one of `Pgr` (pager), `HH` (handheld), or `OEM`, and `Yy` is one of `Mb` (Mobitex) or `Dt` (DataTAC).

When the RIM Wireless Handheld simulator starts, it determines the simulated flash memory size. (The size can be specified on the options dialog or using the `/F` command line option; the default size depends on the handheld being simulated.)

The simulator then checks the current directory for a file named `0sXxxYy.dmp` (where `XxxYy` corresponds to the current simulation platform). If the file exists, its size must be less than or equal to the simulated flash memory size, or the simulator will report an error and abort. If the file does not exist, the RIM Wireless Handheld simulator will create it, and its size is equal to the simulated flash memory size.

Regardless of whether the `.dmp` file existed at startup, the RIM Wireless Handheld simulator always saves the simulated flash memory contents on exit.

If the `.dmp` file is smaller than the flash memory size, it is extended with `0xFF`'s up to the flash memory size. This invalidates any flash memory allocation information present in the file. This facility is provided for use with old `.dmp` files only and should be avoided in normal operation.

Design your algorithms to minimize the impact of the flash memory file system on their performance. The performance of your algorithms will depend on the file system. Therefore, you might have to alter your design so that it is compatible with the handheld's file system.

### Flash memory allocation

The available flash memory (either simulated or real) is divided into four areas:

- file system data area
- unused area
- OS and application code area
- fixed use area

Areas are allocated by writing an entry to a flash memory allocation log, which is a special data structure stored within the fixed use area.

A new entry is added to the log whenever one or both of the `/d` or `/a` command line options are specified. When all of the available log entries have been used, the RIM Wireless Handheld simulator does not allow any more entries to be written and the log is erased.

If the simulator finds no valid log entry in the `dmp` file, a default flash memory allocation is used. To erase the flash memory allocation log, on the **Flash** options tab, click **Erase flash (once)** or use the `/E` command line option.

## Simulating the modem

To model transactions over a radio network, you must provide a modem/network.

You can use either:

- a physical modem (a RAP modem or, on Mobitex networks, the RIM Wireless Handheld)

When a data packet is sent by an application, the packet is sent to the physical modem through the serial port. Packets that are received by the physical modem, as well as status information, are sent through the serial port to the simulator that is running on the computer, and passed on to the applications.

You must have a RAP modem (or RIM Wireless Handheld) with a valid subscription and you must operate in an area that has radio coverage.

- the file system

To send a data packet, the simulator creates a file whose name indicates the destination address. Conversely, the simulator checks for the presence of files whose names indicate that they contain data addressed to the simulator.

This is the default modem method used; it enables you to test the communications functionality of an application without actually connecting to a radio network.

## Using a physical modem

When simulating the radio network using a RAP modem or the RIM Wireless Handheld, the RAP modem becomes the simulator's radio modem.

The simulator is able to communicate with a variety of modems, including all RIM modem products for your network.

### Limitations

The applications are essentially dealing with a live radio network.

- You cannot simulate different coverage situations in software; you are limited by what the modem is actually experiencing. To simulate different coverage situations, change the position of the antenna or obstruct the antenna's coverage.
- The dialog box for manually controlling radio coverage situations is unavailable when running the simulator with a RAP modem, because its behavior is determined by what the actual modem experiences on the live network.

### Options for using a RAP modem

You must set the number of the computer serial port connected to the RAP modem. From the command line, add the `/Rn` option, where `n` is the port number. From the simulator, on the **Configure** menu, click **Options** and click the **Ports** tab; then set the **Serial communications** field to the appropriate port.

### Using the handheld as the modem (Mobitex only)

The RIM Wireless Handheld is an integrated embedded system and radio modem. The radio functionality of the wireless handheld is not normally available through a serial connection.

To allow the RIM Wireless Handheld simulator to communicate with the handheld, load `SDKRadio.dll` (located in the `device\mobitex` folder) onto the handheld to enable it to communicate radio information over the serial port. You can load `SDKRadio.dll` with the `programmer.exe` utility by typing:

```
PROGRAMMER LOAD ..\device\mobitex\SDKRadio.DLL
```

`SDKRadio.dll` automatically detects when the simulator is attempting to contact the handheld, and opens the COM port for communication. When `SDKRadio.dll` is not connected to the simulator, it does nothing. It can coexist with your regular applications on the handheld without harmful effects.



**Note:** The handheld is not designed as a high-capacity modem. After sending large amounts of data, the rate of packet delivery is reduced to save battery life. Sending large amounts of traffic with the handheld can shorten its battery life.

### Using the file system

This simulator offers the follow advantages over using the real network:

- no external hardware is required
- no network account is required
- easy to create coverage situations
- possible to monitor what is sent without the need for extra tools
- easy to write programs to send and receive data packets as a host side

The **Radio simulation control panel** dialog box is available when using the file system to simulate the network. Use it to set coverage conditions.

For information specific to your network, such as the default simulation environment, refer to the *Radio API Reference Guide*.

### Options

Unless you specify otherwise, the simulator stores packets in the current directory, and the **Radio simulation control panel** dialog box does not appear on startup.

You can change the address of the handheld to simulate and the directory in which to store the data packets. Refer to "Simulator options" on page 33 for more information.

## Simulating email

Sending and receiving email through the `message.dll` requires a service book entry; the only way to get a service book entry onto the simulated handheld is to register with RIM server software. In turn, this requires a handheld with an active account.

The easiest way to register your simulated handheld is to use a memory dump from an active handheld of the appropriate type; use the `programmer dump` command.

On Mobitex systems, you can use your handheld as a modem; you can then register the simulated handheld with the RIM server software.

If your application uses the Radio API calls directly, you do not need to register the simulated handheld; registration with the server is only required if your application uses the calls from the Messaging API.

```
C:\dev\app1.dll app2.dll.
```

## Debugging hints

The following tips will help you debug applications using the simulator, and well as using the handheld.

- When the handheld fails and requests a reset, you can often get additional information by typing `debug`. This displays the contents of different registers.
- You might be able to get additional information on a reset request by typing `info`. If the failure was caused by an unresolved OS call, this command displays the name of the call.
- For some reset requests, pressing **R** causes a reset.
- You can force a reset request by pressing **ALT + SHIFT + BACKSPACE**.
- The `RimDebugPrintf()` call writes to the simulator display but has no effect on the actual handheld.

## **Chapter 3: Using the simulator**

# *Chapter 4* **Programming overview**

This section provides information on the following topics:

- Understanding the application environment
- Operating system
- API hierarchy
- Application development steps

## Understanding the application environment

The RIM Wireless Handheld has its own operating system. Many of the features of the operating system, such as the use of flash memory instead of a traditional file system, are based on the requirements of a portable wireless device.

When developing applications for the RIM Wireless Handheld, you should be aware of the following items:

- The RIM Wireless Handheld uses a multitasking operating system, with a co-operative scheduler. Each task must be able to yield so that other tasks can use the CPU. Tasks communicate using MESSAGE structures.
- Tasks and threads are used to run applications.
- The system uses two types of memory, flash memory and RAM. Applications use file system handles to access memory.
- Users interact with handheld applications using the trackwheel and keyboard; users select applications from the handheld Home screen (also called the ribbon).
- Most applications can send email using the Messaging API. If your application needs access to the underlying wireless network, it can use the Radio API.

## Design considerations

When designing and writing applications for the RIM Wireless Handheld, you should keep in the mind the following considerations:

- the trackwheel is the primary input mechanism
- small memory and stack (less than 10 KB)
- limited virtual memory
- co-operative scheduling (each process must yield control appropriately)
- small screen

## Operating system

This section provides an overview of operating system services.

### RIM co-operative scheduler

The handheld's operating system (OS) uses a co-operative multitasking model. Slicing or pre-emption cannot occur between applications. This design removes the need for mutual exclusion mechanisms and semaphores; however, it makes the developer responsible for yielding to other applications during long operations.

Refer to the *Operating System API Reference Guide* for more information.

## Tasks and threads

Each application is created with a single executable thread; new threads can be created and destroyed dynamically.

To create a new task, perform the following tasks:

- Define a `PagerMain()` function.
- Choose an application name.
- Specify an application stack size.

Each `PagerMain()` function is called during the initialization process and is used to construct and initialize objects that the application requires. Refer to "Writing an application" on page 55 for more information.

## Task yielding

Because the operating system uses a co-operative multitasking model, the `PagerMain()` function (and all other code) must include a task yield function call to allow other tasks to make use of the CPU. The application can be restarted in several ways:

- in response to a MESSAGE posted by a handheld system, such as the real time clock
- in response to a MESSAGE posted to the task by another task
- in response the user selecting an icon on the Home screen
- in response to a user selection through the Options List

Refer to "Inter-process communication" on page 50 for more information.

To keep track of task yielding, there are two OS tasks that keep watch on threads:

- **Applications watch**

The applications watch task tracks applications. It triggers a reset (device error 95) when an application runs ten seconds without yielding to another task.

Each time an application spends more than one second without yielding, this is logged. To see the log, press **ALT + SHIFT + B** and select the Watchpuppy line.

- **System watch**

The system watch task monitors the system, and avoids two tasks trading off an application to the exclusion of the rest of the system. It triggers a reset (device error 96) when the system has logged five minutes of system idle time.

## Inter-process communication

The OS enables inter-task communication through posting and receiving MESSAGES. The `RimPostMessage()` and `RimSendMessage()` functions allow a task or system handheld to send a MESSAGE to a specific task or tasks. The `RimGetMessage()` function enables a task to yield control of the CPU until a MESSAGE is posted for it.

The `PagerMain()` function should include code to construct and initialize the necessary application objects, followed by a message loop: an infinite loop containing a `RimGetMessage()` function and code to call the appropriate application function for each expected RIM MESSAGE.

If you are using the UI, a message loop function is included (`Process()`).

Refer to the *Operating System API Reference Guide* for more information.

## Memory use

The handheld contains two types of memory: RAM and flash memory is volatile and can be accessed directly. Flash memory is non-volatile and can be accessed either directly or indirectly through calls to a file system.

On the handheld, code and data are stored in flash. Flash memory can be read as if it were RAM, but with the following two restrictions:

- Flash memory is read-only memory.
- The data might move if the application code yields control of the CPU (by calling either `RimTaskYield(...)` or `RimGetMessage(...)`, by making a file system call, or by calling another application that does one of these things).

Writing to flash memory is relatively slow and requires using special OS commands.

The Database API members are optimized to make the most efficient use of RAM and flash memory. Ideally, you need not be concerned with how the data is stored. File system handles can be assumed to always point to the appropriate data.

## Wireless communications

The BlackBerry SDK provides two different sets of APIs to provide wireless service to applications on the RIM Wireless Handheld

<b>RIM API</b>	<b>Purpose</b>
Messaging API	The Messaging API enables applications to send email messages or other messages that make use of the services offered by the network provider. The Messaging API is independent of the underlying network. Email is always provided as a service.
Radio API	The Radio API provides access to the network at the level of data packets. There are two Radio APIs, one for Mobitex networks and one for DataTAC networks. Because the Radio API is a simplified interface to the network, they have strong similarities.

## User interface

From a hardware perspective, the user interface consists of the screen (LCD), trackwheel, and keyboard. To receive input or display output, an application must become the foreground task.

- Display data on the LCD using OS-level calls or using higher-level objects defined by the UI Engine.
- Whenever a key is pressed, the keyboard handler posts a MESSAGE. The application can acquire the information by getting that MESSAGE through `RimGetMessage()`.

There are two ways for a user to activate an application:

- Select that application's icon on the main screen (ribbon).

To create an icon on the ribbon, register a name and a bitmap using the `RibbonRegisterApplication()` function. The `DEVICE_RIBBON` will then post a MESSAGE to the application's task that can be used to invoke the appropriate application code.

- Select the application in the Options List (**ALT + SHIFT + S**).

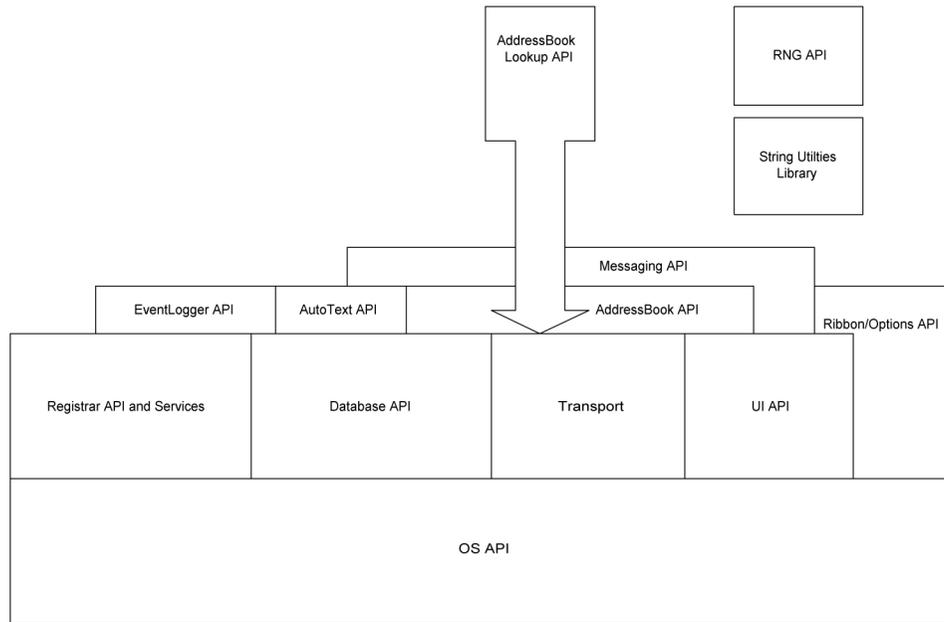
Create an entry in the Options List by registering a name and a callback function using the `OptionsEntry()` function. This function calls the callback function that has access to the LCD (has the foreground) through the Options thread.



**Note:** Use the `OptionsEntry()` approach only to make changes of a limited nature, such as changing the application parameter.

# API hierarchy

The following diagram illustrates the hierarchy of the BlackBerry SDK.



API hierarchy

## Application development steps

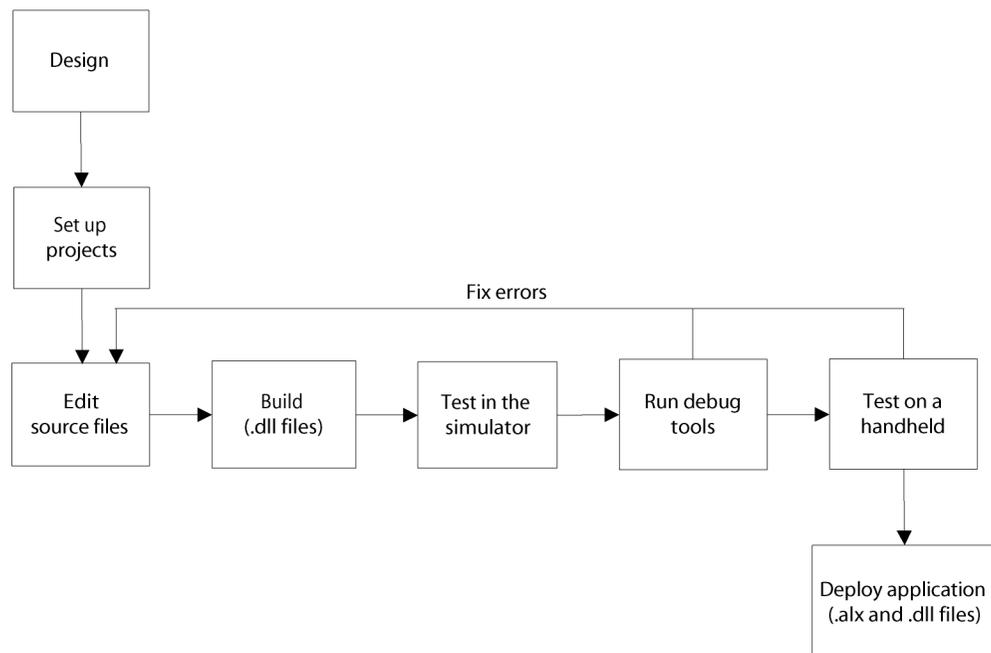
The following steps describe the process for developing an application for the RIM Wireless Handheld.

1. Design your application.
2. In Microsoft Visual Studio, use the application wizard to set up projects for your application and its resources.
3. Edit source files using Microsoft Visual Studio.
4. Build your application as a Windows .dll file.



**Note:** Although the application is compiled as a Windows .dll file, it must not call any Windows-specific code, because it will not run on a Windows system.

5. Test your application using the handheld simulator.
6. Use Microsoft Visual Studio debugging tools to find and correct any problems.
7. Load the application onto an actual handheld for final testing. Refer to "Loading applications for testing" on page 16 for more information.
8. Distribute the application .dll files with application loader (.alx) files for users to load onto their handhelds using the BlackBerry Desktop Manager. Refer to "Deploying applications" on page 25 for more information.



Application development process

# *Chapter 5* **Writing an application**

This section provides information on the following topics:

- Basic program structure
- Defining an entry point
- Registering the application
- Entering the message loop
- Adding the application to the Home screen
- Setting up a basic program structure
- Minimizing memory usage
- Defining version information
- Other example programs

## Basic program structure

The basic program structure consists of an infinite loop that receives messages from the operating system, and code to process those messages.

For every application, you must perform the following tasks:

- create an entry point named `PagerMain()`
- register the application, using a version string and a stack size
- set up a continuous loop that handles system messages

## Defining an entry point

Each handheld application must have an entry point function named `PagerMain` with the following prototype:

```
void PagerMain()
```

The `PagerMain()` function has the format shown in the example on the following page. The following guidelines will help you create the `PagerMain()` function:

- Begin the `PagerMain()` function with the construction of any permanent objects required by the application. You should avoid accessing objects in other `.dll` files because, until the `PagerMain()` functions in these `.dll` files have had a chance to run, the construction of these objects might not have occurred.
- After this local initialization phase, call `RimTaskYield()` to allow the `PagerMain()` functions of other tasks to run.
- After `RimTaskYield()` is called, you can expect that the other tasks' permanent objects have been created. Now call those functions that make the `PagerMain()` function known to other tasks (for example, `RibbonRegisterApplication()`).
- The function can display any user interface elements that are visible to the user if the task is brought to the foreground.
- Finally, if your application needs to respond to `MESSAGES` posted by other tasks or by the system (such as the real-time clock), include an infinite loop containing a `RimGetMessage(...)`. The `RimGetMessage(...)` task yields control of the CPU until a `MESSAGE` is posted for the task.

Refer to the *Operating System API Reference Guide* for more information.

## Program execution life cycle

For C++ programs, global constructors are executed first after a reset. After that, `PagerMain()` is the first part of the program to be executed.



**Note:** Try not to use global object constructors; the objects are permanent and take up valuable resources, resources that could be used by other applications.

`PagerMain()` is called only at startup after a program loads, the handheld resets, or the system stops responding. Turning the handheld on and off does not restart the application at `PagerMain()`. Instead, applications are notified that the pager is turning off by a message sent to them. Because the CPU state, flash memory, and LCD states are preserved when the handheld is turned off, most applications can safely ignore the messages indicating that the handheld is turning off or on.

`PagerMain()` creates and handles the main screen of the program, and effectively enters an infinite loop. The RIM Wireless Handheld assumes all applications to exist. As such, if an application stops itself, either by calling `RimTerminateThread()`, or by returning from `PagerMain()`, the user cannot restart the application directly.

Applications reside continuously in flash memory. Any RAM used by the application is thus also always used. Statically allocated memory cannot be reclaimed by other applications, and dynamically allocated memory (using `malloc()` or C++ operator `new`) can only be re-used by other applications if it is relinquished.

## Registering the application

BlackBerry applications must have two variables that are defined globally:

- version string
- stack size

### Version string

The version string is used to register the application with the task switcher in the operating system. The name appears when the task switcher is run. The definition should be:

```
char VersionPtr[] = "My Application";
```

### Stack size

The stack size is used by the system when it creates the initial thread for the application. The value should be sized according to the needs of your application, although the smaller it is, the better. The variable is `AppStackSize`:

```
int AppStackSize = 5000;
```

The `RimStackUsage()` function provides information on your application's stack usage; it records the highest stack usage since the last call.

When generating a new thread, the `RimCreateThread()` function also enables you to specify a stack size.

## Entering the message loop

If your application uses the UI engine, you can avoid writing a loop by calling `Screen::Process`.

If you need to write a message loop, the typical structure for this loop is a non-terminating loop with a switch statement to delegate message processing to different functions.

The following example demonstrates how to write this type of switch statement.

```
#include "Pager.h"
void PagerMain(void)
{
    MESSAGE msg;
    // perform initialization
    for (;;) {
        RimGetMessage(&msg); // respond to events
        switch(msg.Device) {
            case DEVICE_SYSTEM: // handle SYSTEM messages
                break;
            case DEVICE_TIMER: // handle TIMER messages
                break;
            case DEVICE_KEYPAD: // handle KEYPAD messages
                break;
        }
    }
}
```

You can call `RimGetMessage` from anywhere in the code. For complex programs, every screen of your program should have its own message loop. Messages that do not pertain to the screen, such as radio messages, should be handled by calling a message handler, or even a separate thread.

## Adding the application to the Home screen

Users select handheld applications by selecting icons on the Home screen, which is also called the ribbon.

1. Add `#include <ribbon.h>`.
2. Call the `RibbonRegisterApplication()` function at the beginning of `PagerMain()`, before the program enters the message loop. Refer to the *Ribbon and Options API Reference Guide* for more information.
3. Your application can return to the ribbon by calling `RibbonShowRibbon()`.

Be sure to link with `ribbon.lib`.

---

### Example: PagerMain function

This is a more complicated example of a `PagerMain()` function that includes task yielding and makes use of items from other .dll files.

```
char VersionPtr[] = "Example Application";
int AppStackSize = 5000;

// Function identifying task to the operating system
static void set_task_pid(void)
{
    PID pid;
    pid.Name = "Example Application";
    pid.EnableForeground = true;
    pid.Icon = NULL;
    RimSetPID(&pid);
}

// Function that creates the main thread
void PagerMain(void)
{
    set_task_pid();

    // Yield to let other applications initialize
    RimTaskYield();

    // Register the application with the Ribbon so an
    // icon is shown on the main screen
    RibbonRegisterApplication("Database Sample",&bitmapSample, 0, 0);

    // Initialization complete: set up LCD and enter message loop
    list_screen.Display();
}
```

## Setting up a basic program structure

1. Implement a message loop.

Any application that runs on the handheld must handle messages from the operating system that notify the application of events, such as key presses and radio events. The most efficient way to do this is using a message loop.

The basic structure of the message loop is as follows:

```
for ( ; ; ) {  
    //waits until a message is given to the application by the OS  
    RimGetMessage (&message);  
    //processes message  
}
```

2. To display data on the handheld LCD screen, an application must first become the foreground task.

In this example, the application should be brought to the foreground if the user activates the application by selecting an icon on the main screen. When the user selects the icon, the operating system sends a MESSAGE to the application from the RIBBON device, with the event set to RIBBON\_GRAB\_FOREGROUND. If a RIBBON message is received, you call `RimGetCurrentTaskID()` for the current task handle. You then bring the application to the foreground using `RimRequestForeground()`.

```
if(message.Device == DEVICE_RIBBON) {  
    if(message.Event == RIBBON_GRAB_FOREGROUND) {  
        // Bring application to the foreground so it can use the LCD  
        RimRequestForeground(RimGetCurrentTaskID());  
    }  
}
```

3. If the user presses a key, the operating system sends a MESSAGE to the application from the KEYPAD event. If a KEYPAD message is received, you pass the MESSAGE to the UI Engine, which returns a result code:

```
if(message.Device == DEVICE_KEYPAD) {  
    // Let the UI process the message  
    result = m_ui_engine.HandleInput(message);  
    //Process UI Engine result  
}
```

MESSAGES resulting from a key being pressed on the keyboard are sent only to the foreground task.

4. If the UI Engine returns `CLICKED`, the application should display a menu. If the UI Engine returns `UNHANDLED`, the user might have pressed `BACKSPACE`, in which case the application returns to the Home screen (`RIBBON`).

```
if(result == CLICKED) {
    // Set up, display, and handle menu result
} if(result == UNHANDLED) {
    // Check for the backspace key.
    // If the user pressed backspace, return to Home screen
    if( message.Event == KEY_DOWN && message.SubMsg == KEY_BACKSPACE )
    {
        RibbonShowRibbon();
    }
}
```

## Control flow

A typical application that is running on the RIM Wireless Handheld only has a small LCD, and can only deal with the user doing one thing at time. Because of this, programs can be structured such that any one piece of code only deals with one type of “screen.” When a new screen is entered, it can be implemented as a call to the routine that is handling that screen, and when that screen is exited, the call returns to the previous screen.

In the sample program `filedemo.c` (in `OS_samples\src`), the thread of execution reflects the actual sequence of operations done on the handheld. When the user clicks on the trackwheel, the application calls a routine that contains a message loop to deal with trackwheel input, while the main window does nothing. After the user has made a selection, such as `Add`, a function is invoked, with its own message loop, to compose a new piece of text.

You might want to deal with different types of input at different times. If `RimGetMessage()` is called from multiple places, as is done in the sample program, you should place all the message processing that is the same for all functions into a function, so that it can be called conveniently from all the places from which `RimGetMessage()` is called.

In the case of the `filedemo.c` program, only user input is important, so other messages are ignored.

A sample application is shown below. A more elaborate version of `hello.c` is included with the SDK in the `OS_Samples\src` folder.

### Example: hello.c

```
#include <Pager.h>
#include <ribbon.h>

char VersionPtr[] = "Hello World";
int AppStackSize = 1000;

// Function that creates the main thread
void
PagerMain(void)
{
    // For holding system messages
    MESSAGE msg;
    // Initialize:
    // Yield to let other applications initialize
    RimTaskYield();

    // Register the application with the ribbon
    RibbonRegisterApplication( "Hello &World", NULL, 0, 0 );

    // Enter message loop
    for (;;) {
        RimGetMessage( &msg );
        // respond to events
        switch( msg.Device ) {
            // handle KEYPAD messages
            case DEVICE_KEYPAD:
                // If any key is pressed, return to ribbon
                RibbonShowRibbon();
                break;
            // handle RIBBON messages
            case DEVICE_RIBBON:
                if ( msg.Event == RIBBON_GRAB_FOREGROUND ) {
                    // Bring this application to the foreground
                    RimRequestForeground( RimGetCurrentTaskID() );
                    // Clear screen
                    LcdClearDisplay();
                    // Write string
                    LcdPutStringXY(0,0, "Hello, world!", -1,
                        TEXT_NORMAL);
                }
                break;
        }
    }
}
```

## Minimizing memory usage

The RIM Wireless Handheld is a handheld with limited memory; minimizing memory requirements is an important consideration in designing your applications.

The handheld operating system dynamically allocates stack pages for the applications, which makes it possible to run more applications, but the handheld is now sensitive to the amount of available SRAM and the fragmentation of that SRAM.

Follow these guidelines to minimize your application's memory usage.

### Minimize the amount of SRAM

You can determine how much statically-allocated SRAM an application uses with the programmer `dir` command. Follow these guidelines to minimize the use of SRAM:

- Minimize dynamic memory allocation through `new` and `RimMalloc`.
- Do not allocate many small objects (less than 50 bytes); object overhead is about 12 bytes.
- Try to keep objects below 1 KB in size; use 100 to 1000 bytes each as a guideline.
- Minimize the number and size of static and global variables.
- Verify that any large constant strings, tables, bitmaps, fonts, and other arrays are declared `const` so that they remain in flash memory.

### Free dynamic memory

An application in the background should free as much of its dynamic memory as possible:

- Write long-term state information to flash memory and free as many allocated objects as possible
- If possible, allocate long-term state in a small statically allocated area (that probably contains references to flash memory)

Follow these steps to test for dynamic memory allocation:

1. Check the free memory.
2. Start the application and perform some tasks.
3. Exit to the ribbon.
4. Recheck the memory.

Ideally no memory loss should occur over an application invocation.



**Note:** Running this test on RIM applications might show significant loss the first time that they are entered, because RIM has deferred the allocation of some system buffers until certain applications start.

## Minimize stack space

Minimize the amount of required stack space:

- Avoid placing large objects on the stack so the stack stays below 1 KB.
- If possible, register for Ribbon reactivation and actually exit the thread.
- Minimize the number of threads used by an application.

## Defining version information

The Application Loader that is included with the BlackBerry Desktop Manager looks for version information in the executable code. This version information is inserted by defining two macros in your source module: `RIM_DEFINE_VERSION` and `RIM_DEFINE_DEPENDENCY`.

In addition, `VERSIONINFO` information is used to determine whether an application file is more recent than the installed versions of the file.

The Application Loader uses version information in the following ways:

1. When you attempt to load any applications onto your RIM Wireless Handheld, the loader first attempts to resolve any links between applications; if this fails, the loading process fails.
2. The Application Loader queries the handheld for the list of applications already installed, and their version numbers; it already has a list of applications (and their versions) available on the desktop. The version numbers used here are those stored in the Windows `VERSIONINFO` number, such as 2.0.3.1. For applications without a version number, the loader generates a checksum. The following rules are used to determine the most recent versions:
  - If both files have version numbers, the higher arithmetic value is the most recent.
  - If one file has a version number and the other has a checksum, the one with version information is the most recent.
  - If neither file has a version number and the checksums are different, the version on the desktop is the most recent.

If the version numbers are identical, or neither file has a version quad and the checksums are identical, the files are considered the same and the application is not updated.

3. The Application Loader uses the version information from the `RIM_DEFINE_VERSION` and `RIM_DEFINE_DEPENDENCY` macros to verify that the interfaces in the applications are compatible.

If all steps succeed, the new applications are loaded.

Refer to the Microsoft Visual Studio documentation for information on creating a resource file; only the `VERSIONINFO` field is important for version information.

## RIM\_DEFINE\_VERSION

The RIM\_DEFINE\_VERSION macro has the following syntax:

```
RIM_DEFINE_VERSION(name, major, minor, patch, build, min_compat_maj,
                  min_compat_min)
```

<b>Parameters</b>	<code>name</code>	A string that identifies the module.
	<code>major</code>	Major release number.
	<code>minor</code>	Minor release number.
	<code>patch</code>	Patch number.
	<code>build</code>	Build number.
	<code>min_compat_maj</code>	Earliest major release that is binary compatible with this module.
	<code>min_compat_min</code>	Earliest minor release that is binary compatible with this module.

## RIM\_DEFINE\_DEPENDENCY

The RIM\_DEFINE\_DEPENDENCY macro has the following syntax:

```
RIM_DEFINE_DEPENDENCY(name, min_compat_maj, min_compat_min)
```

<b>Parameters</b>	<code>name</code>	A string that identifies the module.
	<code>min_compat_maj</code>	Earliest major release that is binary compatible with this module.
	<code>min_compat_min</code>	Earliest minor release that is binary compatible with this module.

All modules must have the same dependency. That is, it is acceptable to define these lines in two different modules:

```
RIM_DEFINE_DEPENDENCY(Bar, 1, 5)
RIM_DEFINE_DEPENDENCY(Bar, 1, 5)
```

If there are different definitions, you cannot predict which definition is used by the loader.

You should use macros in a block in the principle public interface header; for example:

```
#ifdef IMPLEMENTING_FOO
    RIM_DEFINE_VERSION(APP_NAME, APP_MAJ, APP_MIN, \
                      APP_PATCH, APP_BUILD, APP_MIN_COMPAT_MAJ, APP_MIN_COMPAT_MIN)
```

## Chapter 5: Writing an application

```
#else
    RIM_DEFINE_DEPENDENCY(APP_NAME, APP_MIN_COMPAT_MAJ, \
        APP_MIN_COMPAT_MIN)
#endif
```

Refer to the `autotext.h` header file for an example.

## Other example programs

The SDK includes several sample applications in the SDK `App_samples` and `OS_Samples` folders. In particular, you might want to look at the file sample program in the `OS_Samples` directory, which consists of `filedemo.c`, `editfunc.c`, and `popupmenu.c`. The program demonstrates how to use the file system and how to provide a graphical user interface for user input.

# ***Chapter 6*** **Operating system services**

This section provides information on the following topics:

- Inter-process communication
- File system services

Refer to the *Operating System API Reference Guide* for additional information.

## Inter-process communication

Any system or application message is in the form of a MESSAGE structure (defined in the header file `Rim.h`). Refer to the *Operating System API Reference Guide* for more information.

### Message passing

Applications on the RIM Wireless Handheld receive external notification through events that are sent to the applications. After an application processes an event, it calls the `RimGetMessage()` function to receive the next event. If no event is available, the application blocks the send process, which enables other applications to run. If no other applications have events to process, the application puts the CPU in a standby state until the next event, such as the expiration of a timer, restarts the application.

`RimGetMessage()` can be called from anywhere within an application. Applications might process messages differently depending on the screen and context of the message. The same application might process the same messages differently depending upon the message context.

To process some events in the same way, regardless of the stage of the application, call `RimRegisterMessageCallback()`. It registers a function to be called for specific types or classes of messages. These functions are called when the main part of the application is blocked on `RimGetMessage()`.

Applications can also exchange messages among their own threads or with other applications. Exchanging messages has an advantage over function calls: when one application calls into another application, the task ID remains the task ID of the calling application. Anything that causes a message to be sent subsequently is sent to the calling application, instead of the called applications.

When sending messages between applications, the `Device` field can be set to one of the system device events. (The macro `DEVICE_USER` can be used for application messages.) If one of the events is specified, the protocol defined here should be maintained.

### Non-blocking and blocking messages

Messages can be sent from one task to another asynchronously or synchronously. In both cases, the receiving task deals with previous events in its message queue before dealing with the new message.

- If the message is sent asynchronously (with `RimPostMessage`), the sending process does not block and immediately continues execution.

- If the message is sent synchronously (with `RimSendMessage`), the sending process blocks until the receiving process processes the event. The sending process does not unblock no more events are in the queue for the receiving process.

When the destination process deals with a synchronous message, it might involve calling other functions (including `RimTaskYield()`). To return information to the sending process, an application can set up a convention of pointing to a specific location. While the receiving process is processing the message, it should place the results in that location. When the sending application unblocks and returns from `RimSendMessage()`, it can use the saved results.



**Note:** Although the OS prevents deadlock from occurring, be careful when sending synchronous messages because tasks are blocked until the receiving application receives the message.

Sending synchronous messages fails if the task ID is invalid or if the destination task is waiting for another task to receive a message. In this case, the `RimSendMessage()` function returns an error immediately.

## Foreground and background

The handheld application server supports multiple threads, as well as multiple applications. Each application on the handheld receives an execution thread at startup. Applications can also create and destroy additional threads dynamically.

Applications co-operatively multitask. At no time can slicing or pre-emption occur between applications. Such a design simplifies application development as it removes the need for mutual exclusion mechanisms and semaphores. However, if a task performs an operation that takes several seconds, it is recommended that the application yield control to other applications periodically so that the handheld does not appear to stop working.

Each thread can run in the foreground or background. A foreground thread can assume control of the display and input; a background thread has no display access. By default, applications are brought to the foreground and threads that are created by `RimCreateThread()` are not.

The display context of the foreground task, including its display bitmap, is shown on the LCD, and receives all keypad and trackwheel input. Each background thread still maintains a copy of the LCD display bitmap in its display context and can manipulate it at any time. When the foreground is changed to a different thread or task, the new task's display bitmap is placed on the LCD, and it now receives all keypad and trackwheel input.

When the foreground is switched from one application to another, the new foreground application receives a `SWITCH_FOREGROUND` message, and the previous foreground application receives a `SWITCH_BACKGROUND` message.

Applications can switch the foreground thread by calling `RimRequestForeground()` to request that a new application be placed on the foreground.

## File system services

The File System API provides a simplified abstraction to the handheld's persistent memory hardware. It protects the persistent data of each application from malicious or careless destruction by other applications.

The handheld file system is different from traditional file systems:

- Non-volatile data is stored in flash memory, using a log-structured file system.
- The file system is presented in database terms. Files are called databases. Database records are equivalent to file records. Records are not a fixed length. File system functions begin with Db.
- You can use streamed file functions, which begin with DbFile, to obtain functionality similar to data streams.
- The file system explicitly provides lookup tables for all of the records in the file system instead of a traditional memory-mapped file system, so that applications still have random access to memory.

## Flash memory and the file system

Non-volatile data in the file system is stored in flash memory. Flash memory provides both read and write operations on a flat memory space.

### Log-structured file system

The file system uses a log-structured file system to make changes to data. Flash memory contents are not rewritten in place. Log file systems have the following key features:

- Changes to data occur sequentially at the end of a continuously growing log.
- When data that already exists in the log is modified, a new copy of the data is written to the end of the log. The old copy of the data is marked as changed but left in the log.
- When the file system runs out of space, old segments of the log are cleaned to make more room for extending the log.

Log file systems are appropriate for the type of usage that is expected on the handheld, where the file system contains many smaller data items and data is typically read much more frequently than it is changed.

### Reading from the file system

To optimize the speed of reading from the file system, the file system provides a rudimentary form of read-only memory mapped access. The handheld file system explicitly provides lookup tables for all of the records in the file system. This

improves efficiency at the cost of slightly more complexity than traditional memory-mapped file access. Applications can read directly out of the flash memory that is used to store the files.

### Random access to files

The handheld file system does not use indexing structures for random access. Searching the file system is performed by sequentially scanning the file from the beginning. Sequential scanning is efficient for short files because the entire file system is in flash memory, which is random access readable.

Applications can obtain even faster random read access to a file by keeping an array of the handles to each record and directly accessing the record through the memory-mapped mechanism.

### Database and streamed file models

The basic abstraction that the file system provides is that of a database—a sequence of variably sized records.

A database can be created, read, modified and deleted. Individual records can be appended at the end of a database, and then read, modified, and deleted. Basic information about databases and records can be retrieved.

Each file consists of a number of separate database entries. Each entry can be nearly 64 KB in size, and can be changed independently. However, the cost of changing a large database entry is a function of its size, so it is not a good idea to have extremely large database entries that change frequently. As well, the file system can only have approximately 6000 handles. Therefore, if the database entries are only 100 bytes in size on average, the handles are exhausted before flash memory storage space is exhausted.

The database also has no concept of files being open or closed. Files remain in memory continuously.

A particular kind of record, called an orphan record, can be created. These records do not belong to a database, but can be added to a database later.

A streamed file enables you to treat a database as a simple sequence of bytes. Functions are provided to open streamed file access to a database, read and modify the data, close the streamed file access and retrieve basic information. An open streamed file is identified by a unique file number, currently represented as an 8-bit unsigned number.

## Memory-mapped file access

Every record and every database in the file system is referenced by a file system handle. Each file system handle uniquely identifies one piece of data in the file system and persists for the life of that data. The address and the corresponding data are located in a record pointer table that is maintained by the file system.



**Warning:** Because records might be moved whenever a portion of the log is cleaned, these pointers are not guaranteed to persist across any calls to the file system that might result in writing to the log (including yielding control to other processes). This applies to all items in the file system, not just the ones to which data is being written.

Applications can safely cache these pointers (for example, in register variables), as long as the values are re-fetched after any operation which might result in a file system write.

Compiler optimizations, such as removing loop invariants from a loop body (which can result in the temporary storage of these pointer values) are typically safe. You are encouraged to explicitly perform such optimization whenever the compiler might fail to do so (and the operations are safe).

The function `DbPointTable` returns the address of the record pointer table. Indexing into this table with the record handle yields a pointer to the actual record. Handles are represented in 16 bits.

For simplicity, the handle table is statically allocated at compile time. As such, the sizing of this table is important, as each potential handle consumes 4 bytes of RAM. The handle table supports about 6000 handles. Because each record has a handle, this limits the system to 6000 records.

### Using pointers to access data

Databases and records are uniquely identified by handles, which persist for the lifetime of the item on the particular system only. Handles are currently represented as 16-bit unsigned numbers.

The file system maintains a system-wide table for mapping handles. The `DbPointTable` function returns the address of the table. Indexing the table with a handle yields a pointer to data. For a database handle, the data represents the database directory entry and, for a record handle, the data is the actual record data. The table currently has 6144 pointers, which limits the number of records to less than 6144.



**Note:** These pointers are not guaranteed to persist across any operation that might result in a change to the file system's permanent data or its organization (such as the cleanup of changed sectors). This includes file system calls and yielding the thread to others applications.

Applications can safely continue to use these pointers until any operation occurs which might result in a change to the file system's permanent state or its organization. Compiler optimizations, such as removing loop-invariant code from a loop body, which might result in the temporary storage of these pointer values, are usually safe. Pointers into flash memory must be re-read from the handle mapping table after writing to the file system or yielding control.

To help detect this error quickly, the RIM Wireless Handheld simulator periodically moves the entire file system, maps the old file system location as invalid, and readjusts the pointer table accordingly. If pointers are copied from the pointer table and reused after yielding to the system, they can subsequently point to regions of memory that have been mapped as invalid. Once these mistakes are trapped, you can find them using Microsoft Developer Studio, because the relevant code causes a page fault.

This behavior can be disabled in the simulator using the `/T-WY` option; however, this only hides these type of errors.

## Using the PointTable edition counter

The OS includes a PointTable edition counter that provides information about the validity of previously stored pointers, including the number of times that the record pointer table changes.

The edition counter is a 32-bit unsigned counter. Its value can be read at any time by calling the `DbPointTableEdition` function.

When the file system is initialized, the edition counter is set to zero, and it increments when a valid pointer within the PointTable changes. For example, it increments whenever a record is deleted or its location is changed by the file system.

After your code performs some changes to the file system and stores some pointers to records locally, it can store the current value of the edition counter.



**Warning:** The edition counter value must be stored *after* changes are complete.

Later, your application can compare the saved edition counter value to the current one. If the values are still the same, stored pointer values are still valid; otherwise, the stored pointer values might no longer be valid and should be discarded.

Keep in mind the following points:

- The edition counter is initialized to zero when the file system is initialized.
- The edition counter value increments by 1 for each file system function call.
- If the edition counter overflows and wraps around, an application with a very old value might incorrectly conclude that the counter's value is the same.

## Chapter 6: Operating system services

- The edition counter values do not persist when file system stops responding or the handheld resets.

Do not use the edition counter before the file system initialization completes successfully. Do not store the edition counter values in files or communicate them externally to the handheld; they have no meaning out of the context of the current handheld code that is running.

---

### Example: Using the PointTable

```
void * *          PointTable;
DWORD *          currentedition;
DWORD           storededition;

PointTable = DbPointTable ();
currentedition = ((DWORD *) PointTable) - 1;

// perform changes to the file system
// get pointers to records and store them locally

storededition = * currentedition;

// do some work, such as change the file system or yield to other
// applications

if (* currentedition == storededition) {
    // stored pointers are still valid;
}

else {
    // stored pointers might no longer be valid;
}
```

## Data security

The file system attempts to maintain security of data through two strategies:

- Most operations that change permanent data are atomic
- Changes to data are verified

### Atomic changes

Most operations that change the permanent data are atomic. For example, record creation and modification operations are atomic. Being atomic means that a record is either completely written and made part of a database, or the database is unchanged. No intermediate state is visible. Data consistency to be preserved if the system stops responding during the function call.

Notable exceptions are the DbAndRec function and streamed file access functions, which are not atomic. Refer to the function descriptions for more information.

## Verification of changes

All changes to the file system's permanent data or its organization are verified by reading the data after it is written completely to flash memory. If any mismatch is detected, it is considered a catastrophic failure, and the system is stopped.



**Note:** Changes made by the DbAndRec function are only partially verified. The system verifies that bits are set to 0 as specified by the data mask, but it does not verify that the remaining bits are unchanged.

## Data portability

This file system uses database and record handles for temporary object identification purposes. Do not store handles (or pointers) in the permanent data. Handles are not portable because they cannot be copied meaningfully to another system.

## File system example

The sample program, which consists of `filedemo.c`, `editfunc.c` and `popupmenu.c`, is demonstrates how to write applications for the RIM Wireless Handheld, including using the file system, user input, and graphical output.

The program implements a program that enables the user to enter and recall short pieces of text. The text is stored in the handheld flash memory file system.

At startup, the sample program calls `DbGetHandle()` to get the handle of its database. If the database does not already exist, an empty database is created by this function call. In either case, the function returns the handle to the database.

Next, the sample program calls its function `GetDatabaseHandles()`. This function uses the `DbFirstRec()` and `DbNextRec()` to collect all the handles to the database, and stores them in an array in RAM. The handles are only 16 bit values, so a short integer array can be used to store the handles to preserve RAM.

After collecting the handles to the database, the program calls `DbPointTable()` to get the address of the handle pointer table. A handle is an index into this array of pointers. This enables the application to access data in the database items without making function calls into the file system.

The following two statements copy a record with a known handle into RAM:

```
Size = DbRecSize(HandleNo, TRUE);
memcpy(EditBuffer, PointerTable[HandleNo], Size);
```

## Chapter 6: Operating system services

Use caution when relying on direct pointer access to the file system. The file system often needs to copy records to modify them. If an application makes calls that write to the file system, or yield to other applications that might write to the file system (for example, by calling `RimGetMessage()` or `Sleep()`), records might not stay in the same place. The application should use values from `DbPointTable()` after any such action.

# *Chapter 7* **Radio communications**

This section provides information on the following topics:

- About the Radio API
- Data packets
- Transmitting packets
- Receiving packets

## About the Radio API

The Radio API provides packet-level access to the radio network. If your application only needs to send messages, such as email or fax, use the Messaging API instead.

The Radio API provides simple API functions to send and receive data. You do not require extensive knowledge of the network to use these function calls, although you will need to understand the basic structure of a data packet: Mobitex Packet (MPAK) on the Mobitex network, or Service Data Unit (SDU) on the DataTAC network.

The APIs for sending and receiving packets on both networks are similar, and the process is identical, even though the names of the APIs and the structures of the data are different.

This chapter provides an overview of how to send and receive packets on both networks. Refer to the *Radio API Reference Guide* for either the Mobitex or DataTAC network for more information.

## Data packets

On the Mobitex network, the operating system can build and format MPAK data packets on behalf of the application. When sending packets over the network, the application fills in the elements of the header structure, and passes it along with the data to be sent to the radio subsystem of the operating system. Alternatively, the application itself can build and format the data packets. The MPAK\_HEADER structure is defined in `mobi tex . h`.

On the DataTAC network, the application must build and format the SDU data packets. The structure of the data packets varies greatly between network providers. The SDU\_HEADER structure is defined in `data tac . h`.

## Transmitting packets

Before sending data packets, an application must register for radio events by calling `RadioRegister()`.

To send a packet, an application must call `RadioSendSdu()` or `RadioSendMpak()` and include the information and length of the data to send. A tag number is returned for each packet submitted to the radio. Once the packet has been transmitted and acknowledged by the radio network, the handheld returns a MESSAGE\_SENT event and tag number to the application that sent the packet.

If the packet could not be delivered to the radio network (for example, the handheld is out of a network coverage area), the application receives the notification message MESSAGE\_NOT\_SENT along with an error code and tag value number. Refer to the *Radio API Reference Guide* for a complete list of errors.

If the packet reaches the network, but the destination is unreachable, the packet is returned to the handheld and is handled like a new received packet.

It is possible to abort transmission of a packet using `RadioCancelSendSdu()` or `RadioCancelSendMpak()`.



**Note:** Cancelling a packet does not guarantee that it was not sent and received before it was cancelled.

The handheld can store internally up to four data packets that are pending for transmission. To send a small number of data packets, an application can call `RadioSendSdu()` or `RadioSendMpak()` repeatedly, without waiting for previous packets to be sent. To send many packets, however, the application must send them one at a time.

## Receiving packets

To receive packets from the network, an application must register to receive radio events by calling `RadioRegister()`.

When packets are received from the radio network, all registered applications are notified by the radio `MESSAGE_RECEIVED` event. Each notified application must retrieve the packet by calling `RadioGetSdu()` or `RadioGetMpak()` before yielding control to the system. After all registered applications receive the message, the packet is released by the system.

If an application cannot store a packet that it receives, the application can return the packet to the system using `RadioStopReception`. `RadioStopReception` enables applications to cancel the reception of packets; however, reception is blocked for all applications, and the modem informs the network that it is no longer ready to receive packets.

The effect of `RadioStopReception()` can be reversed by calling `RadioResumeReception()`. At this time, the stored packets are received again, and the handheld informs the network that it is ready to resume receiving packets.

# *Chapter 8* **Writing UI applications**

This section provides information on the following topics:

- Screens
- Menus
- Fields
- Status boxes
- Dialog boxes
- Frequently Asked Questions

## Screens

A screen is a container for a list of fields, displayed vertically. The UI Engine is responsible for displaying the screen based on the user input and the field that is selected by the user.



Screen

The user can navigate between fields, typically with the trackwheel. A screen can also have a title on the first line of the display that is one line in length.



**Note:** Some fields can increase when the user adds information, and decrease in size when the user deletes information from the field.

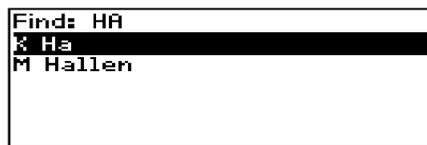
## Screen title

Each screen has a title on the first line of the display. The title line should provide information to users about the current task that they are performing. For example, when a user performs a search in the Address Book application, the screen's first line displays the screen title and the text that the user typed to invoke the search.



Address book screen title

When a user subsequently type **A**, the screen display the records that start with **HA** and the title displays **Find: HA**.



Address book screen

# Menus

Menus are containers for a set of text elements or menu items. Menus appear on top of screens and cover approximately three-quarters of the screen width. Menus should be context-sensitive to the data on the associated screen.



Menu

## Invoking a menu

For consistency, clicking the trackwheel always invokes a menu. To select a menu item, users scroll the trackwheel to select it and then click the trackwheel. A screen is then displayed based on the menu item selected.

## Menu items

The first item on the menu is **Hide Menu**. The UI Engine places this item at the top of the menu automatically. When users click **Hide Menu**, the menu disappears and the previous screen remains on the display.

Another item on the menu is **Cancel**. Users click **Cancel** to stop the current task and return to the previous screen.



**Note:** **Cancel** might not be applicable in all cases. For example, **Cancel** does not appear on the first screen of an application.

## Menu format

A default menu item, which is selected when users open the menu, appears in the middle of the menu. Menu items that are selected more often by users immediately surround the default item; items that are selected less often are further away from the default item.

For example, if the Message List screen is on the LCD display and users click the trackwheel, the following menu appears:



Message List menu



**Note:** The default menu item should not cause irreparable actions, such as loss of data. For example, the user should not be able to delete an entry or a message by clicking the default menu item.

## Fields

Fields are objects that are contained in a screen and which accept user data. Fields take up the width of the display and are typically rectangular.

### Edit boxes

Edit boxes are fields typically used to input text data. An edit box has an optional label (as shown below) and is a field that can grow in size. A cursor is associated with an edit field and word wrapping is performed automatically.



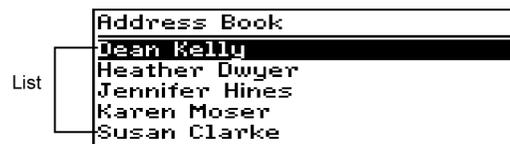
Edit boxes



**Note:** The READ\_ONLY property enables the application to display a text field that cannot be changed by user input.

### Lists

Lists present related items on each line. When a line in the field is selected, the cursor selects the entire line.



List

## Choice boxes

Choice boxes right justify the data, which is either a value in a numeric range or a string that is part of an array of strings.

Choice boxes are actually one-line text boxes that always have a label.



One choice box - Choice boxes always contain one label, in this example the label is In Holster..

Choice boxes

## Separators

Separators separate two fields for readability



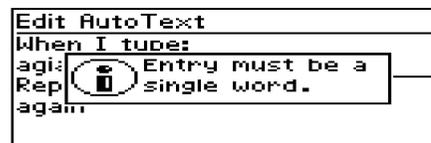
Separator

Separator

Separators are one-line fields that either have a string or bitmap on them. The default string associated with a separator is a line that is the length of the display. Separators can never be selected or highlighted.

## Status boxes

Status boxes are containers for application text or text and bitmaps. A status box appears in the middle of the LCD display and is system modal. It does not receive user input, but the foreground application receives input while the status box is active. The status box can remain active for a period of time that is definable but because it visually disrupts the screen display, it is typically used only when immediate notification is required.



Status box containing application text and a bitmap

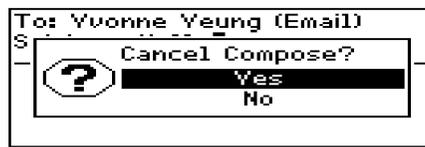
Status box

## Dialog boxes

Dialog boxes are rectangular containers that are application modal and appear in the middle of the LCD display. A dialog box has three components:

- optional bitmap
- text field that displays information about the dialog box
- a list field, edit field, or choice box field for user input

When the dialog box appears, the field for user input is highlighted.



Dialog box



**Note:** The size of the field can be larger than the LCD display.

## Controls

Controls are tools such as icons and buttons that the user selects to issue commands. Keep the number of controls on the display at any one time as small as possible. Too many controls on the display only emphasize the small size of the display.

## Keyboard and trackwheel

As much as possible, try to keep the user from having to move back and forth between the keyboard and the trackwheel. For example, if users are entering data with the keyboard, they should not have to use the trackwheel and then return to the keyboard.

# *Chapter 9* **Messaging**

This section provides information on the following topics:

- About the Messaging API
- Using the Messaging API

## About the Messaging API

The Messaging API is a programming abstraction for creating a message, sending a message, adding attachments to a message, and opening and reading a message. The Messaging API enables developers to use RIM service to send messages. Functionality for receiving messages is restricted to processing attachments only.

Each task in the system can compose only one message at a time. The call to create a message allocates the resources and associates the message with the task. The application adds information to fields, and then queues the message for sending.

### Message structure

A message consists of a list of recipient addresses and a body. The body includes the message text and other optional fields, such as the subject, message ID, and an attachment. This is not the normal division between address and body for an Internet email message, but messages sent on the handheld are not necessarily email messages.

Email messages are always supported, but other types of messages might be supported by the underlying network. The following types of messages are supported by the API: email, fax, phone, one-way page, and two-way page. To support other message types, the network must provide an appropriate message service.

A message body can also contain an attachment, which is an additional file incorporated into the message. Attachments are frequently binary files that must be specially interpreted (or “awakened”) at the receiving end. For example, the **RIM Wireless Handheld** mail software enables users to attach entries from the Address Book database.

### Attachments

An application can send an attachment with a message. It can also register to read attachments. After the user receives email with an attachment and opens the attachment, a callback function that is specified by the registered application is called to process the attachment.

Attachments are added to a message as a byte stream, and are transmitted with the rest of the message. On the receiving side, an application can register with the message application to be notified if a message with a suitable attachment is opened. After the user opens the message, the registered application receives a callback, which enables it to access and process the attachment.

## Using the Messaging API

This section explains how to use the Messaging API function calls to send a message successfully. Sample code is taken from the message application in the `app_samples\Message` directory of the SDK installation.

The code must include the `msg_api.h` header file. When the code is built, it must be linked with the `message.lib` library. The `message.dll` file must be installed on the simulator or the handheld.

There are five steps to sending a message:

1. Create a message (allocate resources).
2. Add recipients (build the recipient list).
3. Add the message text.
4. Add attachment data (optional).
5. Send the message.

### Create a message

This step makes the Messaging application aware of your intention to send a message, and allocates the necessary resources.

This step is just a call to `create_message()`.

```
// Use as return variable for Boolean functions
bool response;

// create the new message
response = create_message();
```

Each task can have one associated message that is under construction. Multiple calls to `create_message()` destroy the existing message and create a new one.

The call fails if sufficient system resources are not available to create a message.

After a task is created, the call `is_composing_message()` returns `true`.

### Adding recipients

The recipient list is built at this stage. All recipients must be added individually. The `field_tag` can be set to one of `TO_FIELD_TAG`, `CC_FIELD_TAG`, or `BCC_FIELD_TAG` to indicate where the recipient is to be placed. (For an email message, the Internet RFC 822 standard requires that an email message have at least one of a TO field or a BCC field.)

Each recipient is added with a call to `add_recipient()`. The call returns a `MsgHandle` to the modified message.

## Chapter 9: Messaging

In some applications, the recipient might be known; in others, the application uses a loop to add recipients when a menu item is selected.

```
// The handle to the current message
DbFieldHandle MsgHandle;

// Add a recipient
MsgHandle = add_recipient(TO_FIELD_TAG, EMAIL,
    "address@nosuch.machine.zcom", "Sample Address");
```

If you must retrieve the field data later, you can use the `get_address_data()` call.

### Adding message text

To build the text of the message, call the `add_field_text()` function. Your application can call the `add_field_text()` function repeatedly; the text that is passed in as a parameter is added to the existing text message.

In other words, you can build your message one sentence at a time by repeated calls to this function, or you can simply set the text pointer to point at an entire paragraph at one time. This flexibility is useful if, for example, your application has to accomplish various amounts of real-time processing with the data as it is being fed into the application.

Even the subject of your message is added using the `add_field_text()` call.

```
// Add the body text to the message
MsgHandle = add_field_text(MESSAGE_FIELD_TAG,
    "Hello, this is ample sample text", 33);
MsgHandle = add_field_text(MESSAGE_FIELD_TAG,
    "for you to use as an example.", 30);
//Give the message a subject
MsgHandle = add_field_text(MESSAGE_SUBJECT_TAG,
    "Some internal rhymes", 21);
```

At this point, the message is finished. You can send it or add an attachment.

### Adding attachment data

Attachments are strung together as a stream of bytes, with no formatting or other processing that is performed by the messaging application. It is the responsibility of your application—the calling application—to ensure that the attachment data is formatted in a manner that is understood by the intended receiving application.

The process for adding an attachment is similar to creating a message:

- Call `create_attachment()` to ask the system to allocate resources for the attachment. The type of the attachment is specified in the parameter `content_string`. (Other parameters have reasonable defaults.)

- Call `add_attachment()` to add data to the attachment. As with message text, you can make multiple calls to `add_attachment()`; new data is added to the existing byte stream.

The actual encoding for the attachment is a compressed version of MIME that is specific to RIM. It assigns byte values to some header texts to shorten the length of the application header for faster transmission. If the message type is `EMAIL`, you should use the strings that are defined for MIME to specify your `content_string` and `content_string_concatenation`. The following sample code demonstrates how to add an attachment:

```
// add a text file attachment to the message
MsgHandle = create_attachment("txt", "",
    "This is the Attachment Text", 27);
```

## Sending the message

A call to `send_message()` causes the messaging application to queue the message for transmission.

The `send_message()` call has a single parameter, `prompt_user_flag`. If this parameter is `true`, the user is prompted to confirm that the message should be sent. If this parameter is `false`, no confirmation is required. In the following sample, confirmation is not required:

```
// Send the message
response = send_message(false);
```

The call queues the message. The system services handle the actual transmission.

## Receiving attachments

When an application desires to receive attachments of a certain type, it should call `register_view_attachment_ex()`. This call specifies the type of attachment the application is interested in and the callback function to be invoked to process the attachment. When the user opens a message which contains an attachment of the desired type, this callback function is called.

The parameters of this callback function are specified in the `register_view_attachment_ex()` function.



**Note:** An application is not awakened automatically as a result of having received a message with the correct attachment type. The callback function is only invoked when the user opens a received message that contains the attachment type.

The calls `get_attachment_data_ex()` and `register_view_attachment_ex()` work together. The calls `get_attachment_data()` and `register_view_attachment()` are deprecated calls that do similar things; do not mix the extended versions with the deprecated versions.

## **Chapter 9: Messaging**

# *Chapter 10* **Ribbon and Options**

This section provides information on the following topics:

- Ribbon
- Options

## Ribbon

The ribbon provides visual cues on how to access the full functionality of the BlackBerry Wireless Handheld. It presents an icon, text hint, and keyboard hotkey for opening applications.

The ribbon library, `ribbon.lib`, also includes the API for setting the wireless handheld's options.

The ribbon consists of a band of icons on the Home screen. The user uses the trackwheel to select an icon, and then clicks the trackwheel to open the application.

When the user clicks an icon, the ribbon sends a message to the application that registered the icon. When the message is received, that application is pushed to the foreground. If the application is not put into the foreground, the ribbon forces it there, in the event that a message was missed or lost.

### Using the Ribbon API

If you are working in Microsoft Developer Studio, on the **Project** menu, click **Settings**. The Settings window appears. Click the **Link** tab. In the **Object/Library Modules** field, add `ribbon.lib` just before `libc.lib`.

To add your application to the ribbon, perform the following steps:

1. Add `#include <ribbon.h>` to the beginning of your program.
2. Add `ribbon.lib` to the list of modules to be linked, before `libc.lib`.
3. Call the `RibbonRegisterApplication()` function at the beginning of `PagerMain()`, before the program enters the message loop.
4. Your application can return to the ribbon by calling `RibbonShowRibbon()`.

Refer to the *Ribbon API Reference Guide* for more information.

### Ribbon messages

All messages that are sent by the ribbon have the Device ID of `DEVICE_RIBBON`. All programming access to the ribbon occurs through the API; no messages are sent directly to the ribbon.

Message	Meaning
<code>RIBBON_GRAB_FOREGROUND</code>	The application has been selected to run. It is placed in the foreground and will start.
<code>RIBBON_HOLSTER_LAUNCH</code>	The handheld has been removed from the holster and the ribbon selects the highest priority application with an active notify request to run.

# Options

The Options application controls the system-wide programming of the handheld features, such as the date and time, screen and keyboard settings, and notification options. In addition, an external API is provided to other applications to centralize their specific options on the main screen. This application, in conjunction with the UI Engine, can be used as a standalone application.

## **Chapter 10: Ribbon and Options**

# Chapter 11

# Database tutorial

This section provides information on the following topics:

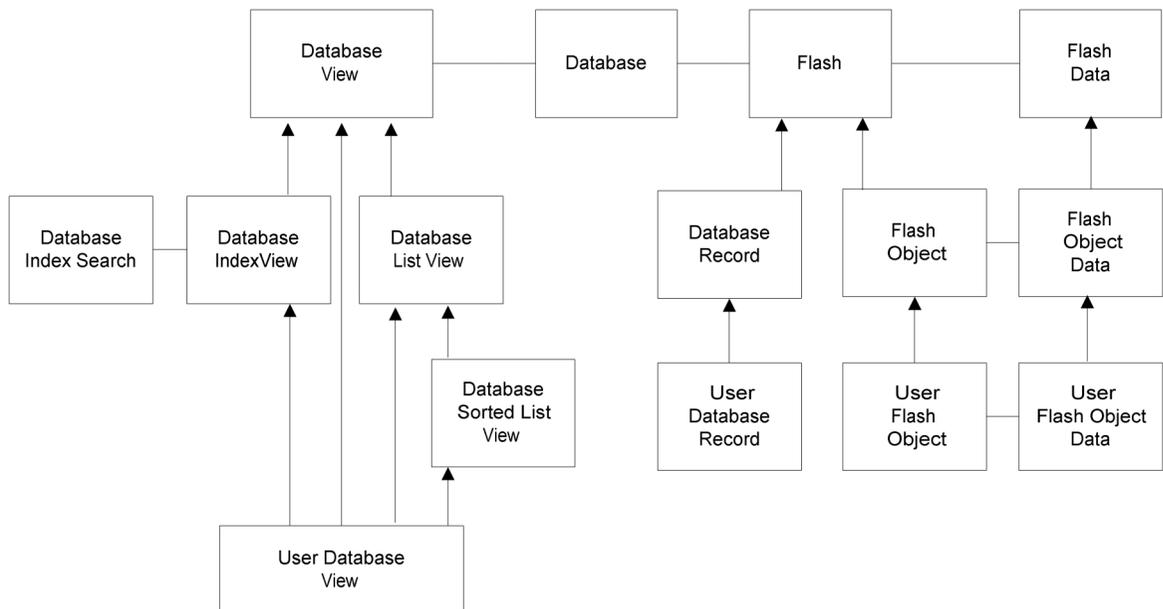
- Database class hierarchy
- Features of the Database API
- Setting up a database
- Storing contacts
- Displaying contacts in a list
- Defining another contact view
- Editing a contact
- Displaying a list using different views
- Updating a list
- UI/Database interaction
- Editing a contact
- Adding an email address
- Saving a contact
- Removing an Email field

This section is intended to give an overview of some of the database's capabilities. Refer to the *Database API Reference Guide* for more information.

## Database API classes

The Database API provides data storage capabilities for applications on the RIM Wireless Handheld. The database library manages flash memory and the file system for the handheld.

The following class hierarchy describes the relationship of the Database, FlashObject, and DatabaseView classes.



Database class hierarchy

### Database

The Database can store multiple FlashObject (or DatabaseRecord) items. The database is responsible for saving data to flash memory, and for updating a list of database views that are attached to it. A Database can be backed up and restored automatically.

### FlashObject

A FlashObject is a fixed-size structure that can store data in a format that is specified by the developer. It can be thought of as a C structure that can be stored in flash memory as well as in RAM. A FlashObject is the most efficient way to store data in a database, but because it does not allow fields to be resized, the size must be fixed at the time of compilation. Each FlashObject must have an associated structure that is inherited from FlashObjectData, where the actual data is stored. The FlashObject can gain access to its data structure by loading the data structure into RAM from flash memory and saving it to flash memory from RAM.

## DatabaseRecord

A `DatabaseRecord` is another class that can store data. Unlike a `FlashObject`, `DatabaseRecord` fields are dynamically sized. As such, they are best used for data with lengths that vary at run time, such as character strings. Fields can also be added to and removed from a `DatabaseRecord` at run time. Each `DatabaseRecord` object can store up to 254 fields at any time, and a field type can be associated with each field.

## DatabaseView

Databases can have up to 16 views attached to them. Each view is updated every time that a record is added, updated or deleted. The following database views are available:

- `DatabaseListView`—This view stores an array of record handles. For example, the message screen uses a `ListView` to store the handles of the message.
- `DatabaseSortedListView`—This view stores a sorted array of record handles.
- `DatabaseIndexView`—This view provides fast lookup of database record handles that match certain keywords, much like an index to a book. For each record, the application can insert several keywords into the index. This index is searchable; a `DatabaseIndexSearch` performs and stores the results of the search.

## PersistentStore

A `PersistentStore` stores a small C-style data structure in flash memory. Example uses include storing user option settings. `PersistentStore` data can be backed up and restored automatically.

## DataBuffer

A `DataBuffer` acts as a simple stream object to store an arbitrary amount of data.

## Bitfield

A `Bitfield` acts as an array of bits. Bit-oriented operations, such as `set` or `clean`, can be performed on individual array entries, or on the array as a whole.

## DbrecordBitField

A `DbrecordBitField` class is derived from the `Bitfield` class. It provides one bit for each possible record handle.

## Database API features

### Data storage

The Database API is well-suited to many applications, especially those that have complex data management concerns. An Address Book application, for instance, can contain hundreds of records that must be sorted quickly, and the user must be able to search through the records without noticing any delay. Applications can define the the format of the data either statically or dynamically, so almost any application data can be stored using the Database API.

### Memory structure

The flash memory on the RIM Wireless Handheld is shared by all applications that are running on the handheld. The flash memory is divided into 64-KB blocks, with a strict limit on the amount of data that can be stored in a single block.

The file system is optimized for handling data stored either as short blocks (less than 80 bytes) or larger blocks (roughly 4 K). If you are designing an application that must store data in blocks larger than 20 KB, you should consider partitioning the data.

The Database API manages these concerns internally on an application's behalf. RAM is used only when necessary, so that other applications will have as much RAM available as possible. Large blocks of data are subdivided internally in many of the data structures, so that finding large contiguous blocks of flash memory space becomes less of a concern.

### Compatibility with the UI Engine

The Database API is designed to work as seamlessly as possible with elements of the UI Engine, such as lists. A database list view, for example, can interact easily with the handle-based design of the UI Engine list. The view manages a subset of the records in a database, and the UI Engine list displays only the handles in that subset.

## Setting up a database

Setting up a database is straightforward. In the following example, you will write a simple Contact Manager application that stores names and associated email addresses.

### To set first name and last name

1. Define a function called `set_name`.

```
bool Contact::set_name(const char * first_name_ptr, const char*
                      last_name_ptr)
{
```

2. Search for a field with the first name tag using `get_first_field_handle`.

```
    DbFieldHandle first_name_handle =
        get_first_field_handle(FIELDTAG_FIRST_NAME);
```

3. Update the field contents with the new name.

```
    // first_name_handle is NULL. A new first name field is created.
    first_name_handle = update_field(first_name_handle,
                                    FIELDTAG_FIRST_NAME,
                                    first_name_ptr, DB_TEXT);
```

4. Repeat steps 2 and 3 for the last name tag.

```
    DbFieldHandle last_name_handle =
        get_first_field_handle(FIELDTAG_LAST_NAME);

    last_name_handle = update_field(last_name_handle,
                                    FIELDTAG_LAST_NAME, last_name_ptr, DB_TEXT);

    return(first_name_handle != DB_INVALID_FIELD_HANDLE &&
           last_name_handle != DB_INVALID_FIELD_HANDLE);
}
```

### To get first name and last name

1. Define a function called `get_name`.

```
bool Contact::get_name(char * & first_name_ptr,
                      const char * & last_name_ptr)
{
```

2. Search for a field with the first name tag using `get_first_field_handle`.

```
    DbFieldHandle first_name_handle =
        get_first_field_handle(FIELDTAG_FIRST_NAME);
```

```
    if(first_name_handle == DB_INVALID_FIELD_HANDLE) {
```

## Chapter 11: Database tutorial

3. If no first name is found in the record, the return value is NULL.

```
        first_name_ptr = NULL;
    }
```

4. Otherwise, return a pointer to the first name of the record using `get_field_data_ptr`.

```
    else {
        first_name_ptr = (char*)
            get_field_data_ptr(first_name_handle);
    }
```

5. Repeat steps 2 through 4 for the `last_name` tag.

```
    DbFieldHandle last_name_handle =
        get_first_field_handle(FIELDTAG_LAST_NAME);
    if(last_name_handle == DB_INVALID_FIELD_HANDLE) {
        last_name_ptr = NULL;
    }
    else {
        last_name_ptr = (char*)
            get_field_data_ptr(last_name_handle);
    }
    return true;
}
```

## To add an email address

1. Define a function called `add_email`.

```
bool Contact::add_email(const char* email_address_ptr,
    DbFieldHandle field_handle)
{
```

2. If `field_handle == DB_INVALID_FIELD_HANDLE`, add a new email field; if it is valid, update the existing field.

```
    DbFieldHandle new_email_handle = update_field(field_handle,
        FIELDTAG_EMAIL_ADDRESS, email_address_ptr, DB_TEXT);
    return(new_email_handle != DB_INVALID_FIELD_HANDLE);
}
```

## To remove an email address

1. Define a function called `remove_email`.

```
bool Contact::remove_email(DbFieldHandle email_field_handle)
{
```

2. Remove the field specified by `email_field_handle`.

```
    return(delete_field(email_field_handle));
}
```

## Storing contacts

In this example, you create a database to store Contact objects temporarily during backup and restore operations.

### To store contact objects

1. Declare a Database object at global scope, and a DatabaseBusyStatus object that the database library will display during any processor-intensive operations, to inform the user that work is in progress.

```
DatabaseBusyStatus ContactBusyStatus;
Database GlobalContactDatabase("Contacts", &ContactBusyStatus)
```

2. Declare a Contact object that the database will use for temporary storage during backup and restore operations.

```
Contact GlobalBRContact(GlobalContactDatabase);
```

3. If the database is to use the backup and restore object that you have created, you must inform the database that it is available by adding the following line to the PagerMain function:

```
GlobalContactDatabase.
    set_backup_restore_object_ptr(&GlobalBRContact);
```

## Displaying contacts in a list

You want to display the contact records in a sorted list. An easy way to sort the contacts would be to have a DatabaseSortedListView. You must define an object that is derived from DatabaseSortedListView, and implement the compare\_objects() function so that the database library knows how to sort the contacts. In this example, contact records are sorted by first name.

### To sort the contacts

1. Define a function called compare\_objects that the database library will call when objects must be sorted.

```
int ContactListView::compare_objects(DbRecordHandle
    contact_1_handle, DbRecordHandle contact_2_handle)
{
```

2. Construct the two objects to be compared (in this case named contact\_1\_handle and contact\_2\_handle).

```
Contact contact_1(GlobalContactDatabase, contact_1_handle);
Contact contact_2(GlobalContactDatabase, contact_2_handle);
```

## Chapter 11: Database tutorial

- Retrieve the names of the contacts to compare using `get_name`.

```
char * first_name_1, * first_name_2;
char * last_name_1, * last_name_2;
contact_1.get_name(first_name_1, last_name_1);
contact_2.get_name(first_name_2, last_name_2);
```

- If the first contact has no first name, and the second contact does, the second contact should be placed first in the list.

```
if(first_name_1 == NULL) {
    // The first contact has no first name
    if(first_name_2 != NULL) {
        return 1;
    }
}
```

- If the first contact has a first name and the second does not, the first contact should be placed first in the list.

```
else {
    if(first_name_2 == NULL) {
        // Return < 0 to indicate the first one should go first.
        return -1;
    }
}
```

- If both contacts have first names, the first names must be compared (if they are equal, sort based on last name).

```
else {
    int strcmp_result = strcmp(first_name_1, first_name_2);
    if (strcmp_result != 0) {
        return(strcmp_result);
    }
}
}
```

- If the contacts have the same first name, sort based on last name in much the same way.

```
if(last_name_1 == NULL) {
    // The first contact has no last name
    if(last_name_2 != NULL) {
        return 1;
    }
}
```

- If both last names are NULL, the two contacts are equal.

```
else {
    return 0;
}
}
```

- If the second contact has no last name, the first contact should appear first. Return a negative value to this.

```
else {
    if(first_name_2 == NULL) {
        return -1;
    }
}
```

10. If both contacts have last names, compare them and return the result.

```
        else {
            return(strcmp(last_name_1, last_name_2));
        }
    }
}
```

## Defining another contact view

In this example, you want to define two views: one that contains all the contacts in the database, and one that contains only the contacts with email addresses.

To decide whether a contact should appear in the email view, you must implement the `is_view_member` function.

### To define the email contact view

1. Determine whether a particular contact should be included with the view of contacts with email addresses.

```
bool ContactEmailListView::
    is_view_member(DbRecordHandle db_record_handle)
{
    Contact contact(GlobalContactDatabase, db_record_handle);
```

2. Search for an email address in the contact record. If there is one, return true, because you want to include this record in the view.

```
    return(contact.get_first_email() != DB_INVALID_FIELD_HANDLE);
}
```

Now, you can define user interface elements that correspond to the database elements you have defined already. Refer to the *UI Engine API Reference Guide* for information on that user interface elements that are used in this example.

## Editing a contact

In this example, you define a class derived from `Edit` called `ContactEdit` that will keep track of a field handle and a field tag for the `Edit` field. That way, when the user enters or edits a contact, the application knows what type of data should be stored in the field. When the user attempts to remove an email address, for example, the `ContactEdit` field knows which field handle must be removed from the corresponding `Contact`.

### To associate an edit field with a record

1. Construct the `Edit` buffer. The data in each field can have a maximum length as specified by the constant `MAXLEN_CONTACT_DATA`.

```

ContactEdit(const char* label_ptr,
            DbFieldTag tag = DB_INVALID_FIELD_TAG,
            DbFieldHandle handle = DB_INVALID_FIELD_HANDLE)
: Edit(label_ptr, MAXLEN_CONTACT_DATA,
      MAXLEN_CONTACT_DATA),
  m_field_handle(handle), m_field_tag(tag)
{}

```

2. Return the field handle that is associated with this edit field.

```

DbFieldHandle get_field_handle()
{
    return(m_field_handle);
}

```

3. Return the field tag that is associated with this edit field.

```

DbFieldTag get_field_tag()
{
    return(m_field_tag);
}
protected:
DbFieldTag    m_field_tag;
DbFieldHandle m_field_handle;
};

```

You also need a `ContactList` object, which is derived from the UI Engine `List` object. You want to be able to display a list for either one of the views: the view of all `Contacts`, and the view of just the `Contacts` with email addresses. The `List` object and the `DatabaseListView` object are designed to work together easily.

## Displaying a list using different views

To display a particular record on the LCD, UI Engine calls the `NowDisplaying` function, passing in the index to the list that corresponds with that record. The `NowDisplaying` function must convert the index to a record handle, so you need a `get_handle` member function.

This function determine whether all contacts or only email contacts are being displayed, and returns a handle from the appropriate view.

```
DbRecordHandle ContactList::get_handle(int index)
{
    if(m_show_all_contacts_flag) {
        return(m_all_view[ index ]);
    }
    else {
        return(m_email_view[ index ]);
    }
}
```

Now you can define the `NowDisplaying` function. Its main task is to convert a database record into a string that can be displayed on the LCD.

### To display <first name> <last name>

1. Define a function called `NowDisplaying`.

```
void ContactList::NowDisplaying(const int index)
{
```

2. Get the record handle of the contact to be displayed.

```
    DbRecordHandle db_handle_to_display = get_handle(index);
```

3. Construct the appropriate contact object.

```
    if(db_handle_to_display != DB_NULL_HANDLE)
    {
        Contact contact(GlobalContactDatabase, db_handle_to_display);
```

4. Get the first and last names of the contact, and use `RimSprintf()` to combine them into a single string.

```
        char full_name_buffer[ MAXLEN_CONTACT_DATA * 2 ];
        char * first_name_ptr, * last_name_ptr;
        contact.get_name(first_name_ptr, last_name_ptr);
        RimSprintf(full_name_buffer, sizeof(full_name_buffer),
            "%s%s",
            first_name_ptr ? first_name_ptr : "",
            first_name_ptr && last_name_ptr ? " " : "",
            last_name_ptr ? last_name_ptr : "");
```

5. Tell the UI Engine to display the string.

```
        PutColumn(full_name_buffer);
    }
}
```

## Updating a list

When the number of entries in the list changes (for example, when you add or remove an item from the view, or when you decide to show email contacts instead of all contacts), you must update the UI Engine with the new number of entries in the view.

### To update a list

1. Define a function called `update_num_entries`.

```
void ContactList::update_num_entries()
{
```

2. Depending on whether you are showing all contacts or just email contacts, tell the UI Engine how many handles are in the appropriate view.

```
    if(m_show_all_contacts_flag) {
        SetNumEntries(m_all_view.get_num_handles());
    }
    else {
        SetNumEntries(m_email_view.get_num_handles());
    }
}
```

3. Tell the UI Engine to Redraw the visible portion of the list in case it has changed.

```
    Redraw();
}
```

### To switch between the views

1. Define a function called `show_all`.

```
void ContactList::show_all(bool show_all_contacts)
{
```

2. Set the `show_all_contacts_flag` appropriately.

```
    m_show_all_contacts_flag = show_all_contacts;
```

3. Update the UI Engine and the display.

```
    update_num_entries();
}
```

## UI/Database interaction

To convert a selected item in the list to a database record handle, use the following code:

```
list_index = m_contact_list.GetSelectedIndex();
db_handle = m_contact_list.get_handle(list_index);
```

The contact list screen has menu items that enable the user to switch between the view of all contacts and email contacts.

### To switch between two different views

1. If the user selects the **Show all contacts** menu item, update the list accordingly.

```
case CONTACT_LIST_MENU_SHOWALL:
    // Show all of the contacts
    m_contact_list.show_all();
    m_title.SetText("All Contacts");
    break;
```

2. If the user selects the **Show Email contacts** menu item, update the list accordingly.

```
case CONTACT_LIST_MENU_SHOWEMAIL:
    // Show only the contacts with an email address
    m_contact_list.show_all( false );
    m_title.SetText("Email Contacts");
    break;
```

## Editing a contact

When the user wants to edit a contact, you need to set up the screen to show the information that already exists for that contact.

### To display a contact's information

1. Construct a Contact object to modify.

```
Contact contact(GlobalContactDatabase);
```

2. Reconstruct the Contact object so that the user can edit it.

```
if(db_record_handle != DB_NULL_HANDLE) {
    contact.reconstruct(db_record_handle);
}
```

3. Populate the name fields on the screen.

```
char * first_name_ptr, * last_name_ptr;
contact.get_name(first_name_ptr, last_name_ptr);
```

## Chapter 11: Database tutorial

```
// Place the names in the name fields
if(first_name_ptr) {
    m_first_name_edit.Insert(first_name_ptr);
}
if(last_name_ptr) {
    m_last_name_edit.Insert(last_name_ptr);
}
```

4. Enumerate the email fields, and place each one on the screen.  

```
DbFieldHandle email_field_handle = contact.get_first_email();
while(email_field_handle != DB_INVALID_FIELD_HANDLE) {
```

5. Construct a new contact field to hold the email address.

```
ContactEdit* contact_edit_ptr = new ContactEdit("Email:",
    Contact::FIELDTAG_EMAIL_ADDRESS, email_field_handle);
```

6. Set the properties of the edit field so that it handles email addresses in an effective way.

```
if(contact_edit_ptr) {
    contact_edit_ptr->AddProperties(Edit::CR_TO_ROLL_DOWN |
        Edit::EMAIL_FIELD);
}
```

7. Insert the field contents into the edit field.

```
contact_edit_ptr->Insert((char * const)
    contact.get_field_data_ptr(email_field_handle));
}
```

8. Find the next email field. If one exists, proceed to step 5. If not, the screen is complete.

```
        email_field_handle =
            contact.get_next_email(email_field_handle);
    }
}
```

## Adding an email address

To add an email address to the contact, you add a `ContactEdit` field to the screen with a field handle of `DB_INVALID_FIELD_HANDLE`, to indicate that this particular email address is not saved. When the contact is saved, the email addresses on the screen are saved with the contact.

### To add an email address

1. Create a new field and add it to the screen.

```
case CONTACT_EDIT_MENU_ADDEMAIL:
{
    ContactEdit* new_email_ptr = new ContactEdit("Email:",
        Contact::FIELDTAG_EMAIL_ADDRESS);
```

2. Set the properties of the email field so that it handles email addresses in an effective way.

```
if(new_email_ptr) {
    new_email_ptr->AddProperties(Edit::CR_TO_ROLL_DOWN |
        Edit::EMAIL_FIELD );
    AddField(*new_email_ptr);
}
```

3. Set the focus to the new field.

```
    SetFieldwithFocus(new_email_ptr);
}
break;
}
```

## Saving a contact

When the user decides to save the contact, you need to extract the contact information from the fields that are on the screen.

### To save a contact

1. If the user selects the **Save contact** menu item, enter the first and last names in the edit fields.

```
case CONTACT_EDIT_MENU_SAVE:
{
    const char * first_name_ptr = m_first_name_edit.GetBuffer();
    const char * last_name_ptr = m_last_name_edit.GetBuffer();
}
```

2. If the user attempts to save a contact without entering a name, bring up a dialog box to notify the user, and do not save the record.

```
if(*first_name_ptr == '\0' && *last_name_ptr == '\0') {
    OKDialog ok_dialog("The contact must have a name.");
    ok_dialog.Go(m_ui_engine);
    break;
}
```

3. Set the names for the contact.

```
contact.set_name(first_name_ptr, last_name_ptr);
```

4. Go through all of the fields on the screen, and add the email addresses that are stored in the email fields.

```
Field* field_ptr = GetFirstField();
while(field_ptr != NULL) {
```

5. If the field is either the first\_name or last\_name field, it does not contain an email address, so ignore it.

```
if(field_ptr != &m_first_name_edit && field_ptr !=
    &m_last_name_edit) {
```

## Chapter 11: Database tutorial

- When you arrive at an email field, add the address.

```
ContactEdit* contact_edit = (ContactEdit*) field_ptr;
const char * email_address = contact_edit->GetBuffer();
```

- If the email\_address field is not empty, add or update the address.

```
if(email_address != NULL && *email_address != '\0') {
    contact.add_email(email_address,
        contact_edit->get_field_handle());
}
}
```

- Retrieve the next field on the screen. If one exists, proceed to step 5. If not, the email addresses have all been added to the record.

```
field_ptr = GetNextField(field_ptr);
}
```

- Save the record with all of the fields.

```
contact.save();
return;
}
```

## Removing an email field

If the cursor is positioned over an email field, you want to provide a menu item that enables the user to remove the email field.

### To display the Remove Email menu item

- Retrieve the field where the cursor is currently positioned.

```
Field* current_field_ptr = GetFieldwithFocus();
```

- If it is neither the first or the last name, it must be an email address, so enable the **Remove Email** menu item.

```
if(current_field_ptr != &m_first_name_edit &&
    current_field_ptr != &m_last_name_edit) {
    m_menu.ShowItem(CONTACT_EDIT_MENU_REMOVEEMAIL);
}
```

- If it is the first or last name field, do not show the menu item.

```
else {
    m_menu.HideItem(CONTACT_EDIT_MENU_REMOVEEMAIL);
}
```

- Set the default menu item to **Save**.

```
m_menu.SetSelectedIndex(CONTACT_EDIT_MENU_SAVE);
```

If the user selects the **Remove Email** menu item, you must remove the email address.

If the `ContactEdit` field has a valid handle, the email address has already been saved with the record, so you need to remove it from the record and the LCD.

If the field handle is invalid, the email address is not saved, so you only need to remove it from the display.

## To remove an email address from the record

1. If the address is saved in the record, it has a valid field handle, so you want to remove it from the record.

```
case CONTACT_EDIT_MENU_REMOVEEMAIL:
{
    ContactEdit* contact_edit = (ContactEdit*) current_field_ptr;
    DbFieldHandle field_handle =
        contact_edit->get_field_handle();
    if(field_handle != DB_INVALID_FIELD_HANDLE) {
        contact.remove_email(field_handle);
    }
}
```

2. In either case, you need to remove the email address from the display.

```
RemoveField(*contact_edit);
break;
}
```

## **Chapter 11: Database tutorial**

# ***Chapter 12***

# **Bitmaps, fonts, and sounds**

This section provides information on the following topics:

- Creating bitmaps
- Converting existing bitmaps
- Creating custom fonts
- Resource .dll files

## Creating bitmaps

You can use the `bitmap.exe` utility to convert a Windows bitmap file into a bitmap that can be displayed on the handheld. This utility supports any color depth.

Bitmaps in the form of graphics and custom icons can be created and displayed on the LCD. You can also copy the handheld screen to a buffer in flash memory for future recall. This is especially useful for preserving displays when switching between multiple applications that write to the screen.

The maximum size for a bitmap is 256 by 256 pixels. If graphics are larger than this size, the `bitmap.exe` utility does not scale the image; it truncates the image and displays the portion at the top right.

Follow these steps to create bitmaps:

1. Create a definition file. The definition file, which has the `.def` extension, defines variables as well as the arrays that make up the bitmap:

- `BITMAP_NAME`—specifies a unique name for the bitmap header
- `STATIC`—set to either `ON` or `OFF`

A value of `OFF` specifies that the bitmap to be global to all modules. A value of `ON` specifies that the bitmap is visible only to the module or function in which it is defined.

- `type`—must be 0 to indicate a monochrome bitmap
- `height`—height of the bitmap, in decimal
- `width`—width of the bitmap, in decimal
- `end`—specifies the end of a bitmap

For example, here are the contents for a file called `face.def`:

```
#define BITMAP_NAME face
STATIC OFF
TYPE=0
WIDTH=16
HEIGHT=16
=
[      xxxx      ]
[   xxx   xxx   ]
[  x       x   ]
[ x       x   ]
[ x       x   ]
[ x  xx   xx  x ]
[x  xx   xx  x ]
[x       x   ]
[x       x   ]
[x       x   ]
[ x x       x x ]
[ x  xx   xx  x ]
[ x   xxxx  x   ]
[  x       x   ]
```

```
[   xxx   xxx ]
[      xxxx ]
END
```

Multiple bitmap definitions can be placed within a single bitmap definition file.

Any non-space character within the data is considered black. A space is white.

Comment lines (/) and preprocessor commands (#) are passed through to `face.h`. Assembler-style comments (;) are ignored. Lines beginning with spaces or tabs are ignored as well.

2. Use the `bitmap.exe` utility to convert the bitmap definition into a C header file that can be included with source code. For our example `face.def` file, the command line would be:

```
bitmap face.def face.h [options]
```

- Options**
- mono This option converts monochromatic images with simple thresholding; recommended for text.
  - sf This option specifies to use Steinberg Floyd (diffusion) conversion to convert to monochromatic color depth; recommended for photographs.

3. Include the bitmap with the C source code. To display the bitmap in your code, use the `LcdCopyBitmapToDisplay` function.

The code to use `face.h` might look like this:

```
#include "FACE.H"
.
.
.
LcdCopyBitmapToDisplay( &face, 58, 8);
```

Result:



## Converting existing bitmaps

You can use the `BMP2DEF.EXE` utility to convert existing bitmap files (`.bmp`) into definition files. You can then use the `bitmap.exe` utility to convert the definition file into a bitmap for the handheld. Refer to "Creating bitmaps" on page 116 for more information.

1. Convert the bitmap file (`.bmp`) into a definition file (`.def`). For example:

```
bmp2def pgrlogo.bmp pgrlogo.def
```

## Chapter 12: Bitmaps, fonts, and sounds

2. Convert the bitmap definition (.def) file into a C header file (.h) so that it is included with the source code. You can do this by using the `bitmap.exe` utility that is included with the SDK.

```
bitmap pgrlogo.def pgrlogo.h
```

3. Open the newly created header file and give the `bitmap` structure a name (for example, `Logo`).
4. Include the bitmap with the C source code. The `LcdCopyBitmapToDisplay` function enables you to display the bitmap.

```
#include "pgrlogo.h"  
.  
.  
.  
LcdCopyBitmapToDisplay( &logo, 58, 8);
```

There are some restrictions on the size and type of bitmap that can be loaded:

- Bitmaps must be monochrome. Grayscale or color bitmaps are not supported.
- Bitmaps larger than 132-by-65 pixels will be truncated to 132-by-65 pixels. Bitmaps taller than 65 pixels will be missing lines from the top, and bitmaps wider than 132 pixels will be missing columns from the right.
- Bitmaps larger than 256 by 256 pixels are not supported.
- Compressed bitmaps are not supported.
- Bitmaps files (.bmp) created by OS/2 are not supported.

## Creating custom fonts

The BlackBerry SDK includes two built-in fonts: 8-line normal and 6-line normal. In addition to these, you are free to create your own fonts, which are stored in application code space. (Created fonts are often stored in a resource .dll file. Refer to "Creating a resource font .dll file" on page 121 for more information.)

In addition to displaying individual typesets, custom fonts provide an added degree of versatility for applications that use icons. For example, you might choose to display frequently used icons as symbols from a custom font, rather than as bitmaps, when the icons are to appear aligned on a baseline with text. Fonts can be either proportional or fixed-width.

The process for creating custom fonts is similar to the process for creating bitmaps.

There is no restriction on the number of different fonts that can be displayed simultaneously. Each display context has five fonts associated with it, which are numbered 0 to 4. The default fonts are 8-line normal and 6-line normal.

You can easily incorporate custom fonts by following these steps.

1. Create a definition file for the font. The definition file should have the extension .def and should contain definitions and values for the following items:

## Creating custom fonts

- `FONT_NAME`—specifies a unique name for the font header
- `NAME`—specifies the external name of the font (currently not used)
- `FIRST` and `LAST`—specifies the first and last characters of the font (they can be written as the actual character enclosed in apostrophes or written in hexadecimal format)
- `HEIGHT`—specifies the fixed height of the font, in decimal format
- `PROPORTIONAL`— specifies that each character has proportional widths (for fixed width characters, use `WIDTH` instead)
- `WIDTH`—if `PROPORTIONAL` is not included, specifies the fixed character width, in decimal format
- `underline`—specifies, in decimal format, the row from the top on which an underline should appear
- `end`—specifies the end of the font definition

There are several ways to start a new character.

Example	Description
<code>= ('A')</code>	This defines a specific character.
<code>=0x7F</code>	This defines an ASCII character using hexadecimal format
<code>==+</code>	This defines the next ASCII character
<code>== -</code>	This defines the previous ASCII character

Any non-space character within the data is considered black. A space is white.

Comments lines (`/`) and preprocessor commands (`#`) are passed through to `SCRIPT.H`. Assembler style comments (`;`) are ignored. Lines beginning with spaces or tabs are ignored as well.

For example, here is a text file called `SCRIPT.DEF`:

```
#define FONT_NAME script
NAME=Script
FIRST=' '
LAST=0x7F
HEIGHT=7
PROPORTIONAL
UNDERLINE=7
;; no space between '=' and character
=' '
[ ]
[ ]
[ ]
[ ]
[ ]
[ ]
[ ]
```

## Chapter 12: Bitmaps, fonts, and sounds

```
= '!'  
[x ]  
[x ]  
[x ]  
[x ]  
[ ]  
[x ]  
[ ]  
; advance to next character  
=++  
[x x ]  
[x x ]  
[ ]  
[ ]  
[ ]  
[ ]  
=++  
[ x x ]  
[xxxxx]  
[ x x ]  
[xxxxx]  
[ x x ]  
[ ]  
[ ]  
END
```

2. Use the `lcdfonts.exe` utility to convert the font definition into a C header file to be included with source code in a resource .dll file or an application.

```
lcdfonts script.def script.h
```

3. Include the custom font with the C source code. The `LcdReplaceFont()` and `LcdSetCurrentFont()` functions are used to set fonts in the current context.

```
#include "SCRIPT.H"
```

```
if (LcdReplaceFont (0, &script) == LCD_OK)  
    LcdSetCurrentFont(0)
```

## Resource .dll files

Resource .dll files contain resources such as fonts and sounds but do not contain executable code. When you download a resource .dll file onto the handheld, the operating system detects it and keeps a record of all the resources it contains. You can then access those resources using the API that is provided by the operating system.

## Creating a resource .dll file

Generally, to create a resource .dll file, you create a new C file, compile it, and then link it with `ResEntry.obj`.

For any resource .dll file, the general procedure is as follows:

1. In the C source file for the .dll file, assign a name for the resource to the variable `char *DLLName`.
2. Include the header file `RESOURCE.H`.
3. Include header files that are specific to your resource, such as `ResourceFonts.h` for a font resource, `Song.h` for a tunes resource, and the header files defining resource itself.
4. Declare other structures or instances of structures as required by the resource.
5. Declare an instance of `ResourceStruct`. The `ResourceStruct` accepts the following arguments:
  - type of resource in the .dll file, either `RESOURCE_FONT` for a font or `RESOURCE_SONG` for a tune.
  - number of elements in the resource.
  - third argument is always 0.
  - addresses of the elements defined above.

Follow these guidelines for writing the main C file for a resource .dll file:

- The file must include the header file `RESOURCES.H` and the header file specific to the resource that you are using.
- The file should contain a `PagerMain()` function.

Build the .dll file as you would any RIM application, except link it with `ResEntry.obj` instead of `OsEntry.obj`. Do *not* link `OsEntry.obj` with a resource .dll file as this file is only used with executable applications.

More detailed descriptions for creating font and tunes .dll files appear below.

## Creating a resource font .dll file

1. Create a C file and define a name for this .dll file as follows:

```
char *DLLName = "FontD11";
```

2. Include the following header files:

```
#include "resources.h"
#include "ResourceFonts.h"
```

3. Include the header files for the fonts that you want to include in your font .dll file.

For example, if you have created two font header files (using `LCDFONTS.EXE`) for Arial and Times at 8-points high, include this code:

```
#include "arial8.h"
#include "times8.h"
```

4. Declare an instance of `ResourceFontStruct` for each font. Each instance contains the font name and a pointer to the actual font, which is defined in the font header files.

```
ResourceFontStruct Font1 = ("Time", &Time);
ResourceFontStruct Font2 = ("Arial" &Arial);
```

## Chapter 12: Bitmaps, fonts, and sounds

5. Declare an instance of `ResourceStruct`. This accepts the following arguments:
  - type of resource in this .dll file (use `resource_font` for fonts)
  - number of elements in that resource (in this example, there are 2 fonts)
  - the third argument is always 0
  - addresses of all font structures defined above

The following sample demonstrates how to create a `ResourceStruct`:

```
# define NUM_FONTS 2
ResourceStruct ResStruct = {
    RESOURCE_FONT,
    NUM_FONTS,
    0,
    { (void*) &Font1, (void*) &Font2 }
};
```

### Testing the font .dll file

The following code sample demonstrates the use of resource fonts. The code determines the total number of fonts that are installed using `LcdGetNumberOfFonts()`. It then loops and prints the name of each font in that font. The font name is obtained through a call to `LcdGetFontName()`.

```
const char      *      fontName;
int             numFonts;
int             screenOffset = 8;

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

//For each font, make it the current one, and then print its name
for (i = 0; i < numFonts ; i++) {
    LcdSetCurrentFont(i);
    if (LcdGetFontName(i, &fontName) == LCD_OK){
        //Get name
        //Display font name
        LcdPutStringXY(0, i * screenOffset, fontName, -1, TEXT_NORMAL);
    }
}
```

### Creating a resource tunes .dll file

1. Create a C file that defines a name for this .dll file as follows:

```
char *DLLName = "SongD11";
```
2. Include the following system header files:

```
#include "resources.h"
```

```
#include "Song.h"
```

The file `song.h` contains declarations for the `NOTE` and `SONG` data structures that are used to define any tunes that are used on the system.

3. Declare an array of `NOTE`s that you want to play. Each note is a frequency-duration pair, for example:

```
const NOTE MySongNotes [] = {
    440, 200,
    550, 100,
    440, 100,
    550, 150,
    330, 300,
    440, 400
};
const NOTE MyOtherSongNotes [] = {
    3000, 100,
    2000, 100,
    1000, 100,
    2000, 100,
    3000, 100,
    2000, 100,
    1000, 100,
    2000, 100,
};
```

4. Define your tune as a `SONG` structure. A `SONG` contains three fields:
  - number of notes in the tune (which can be determined by using the `length_of` macro)
  - name that you want to call the tune
  - a pointer to the `NOTE` array that contains the notes that define the tune

For example:

```
const SONG MySong = {
    LENGTH_OF (MySongNotes), "My song", MySongNotes};
const SONG MyOtherSong = {
    LENGTH_OF (MyOtherSongNotes), "My song 2",
MyOtherSongNotes
};
```

5. Declare an instance of `ResourceStruct`. This takes the following arguments:
  - type of resource in this .dll file (use `resource_SONGs` for tunes)
  - number of elements in that resource (in this example, there are 2 tunes)
  - the third argument is always 0
  - addresses of all song structures defined above

For example:

## Chapter 12: Bitmaps, fonts, and sounds

```
ResourceStruct ResStruct = {
    RESOURCE_SONGS,
    NUM_SONGS,
    0,
    {
        (void*) &MySong,
        (void*) &MyOtherSong
    }
};
```

### Testing the tunes .dll file

The following code sample demonstrates the use of resource tunes. The code determines the total number of tunes that are installed using `RimGetNumberOfTunes()`. It then loops and both plays and prints the name of each tune. The code then retrieves the name of each tune through a call to `RimGetTuneName()`.

```
const char *      tuneName;
int               screenOffset = 8;
int               numTunes;

//Obtain the total number of available tunes
numTunes = RimGetNumberOfTunes();

for (i = 1; i < numTunes ; i++ ) {
    RimAlertNotify ( i, 1 );      //play the tune
    RimSleep ( 50 );             //pause for a bit
    if (RimGetTuneName ( i, &tuneName ) == TUNE_OK) {
        //Display tune name
        LcdPutStringXY (0, (i - 1) * screenOffset,
            tuneName, -1, TEXT_NORMAL );
    }
}
```

# *Chapter 13* **C library compatibility**

This section provides information on the following topics:

- Summary of C compatibility
- Compatible functions
- Functions that are not compatible

## Summary of C compatibility

Only some functions in the compiler C library are safe to call from the RIM Wireless Handheld environment. The following table summarizes the information.

Function Group	Compatible with RIM Wireless Handheld?		
	Yes	No	Some
Argument access macros	●		
Buffer access macros	●		
Byte classification		●	
Character classification		●	
Data conversion			●
Debug		●	
Directory control		●	
Error and exception handling		●	
File handling		●	
Floating point			●
Input/output		●	
Locale dependent		●	
Memory allocation		●	
Process and environment control		●	
Searching and sorting	●		
String manipulation		●	
System calls		●	
Time		●	

## Compatible functions

This section explains C library functions that are compatible with the RIM Wireless Handheld.

### Argument access macros

This set of functions is compatible with the RIM Wireless Handheld environment. This includes the macros `va_arg`, `va_start`, and `va_end`.

### Buffer manipulation functions

This set of functions is compatible with the RIM Wireless Handheld environment. This includes `memcpy`, `memchr`, `memcmp`, `memcpy`, `_memcpy`, `memmove`, `memset`, and `_swap`.

### Data conversion functions

The following data conversion functions can be used in a RIM Wireless Handheld application: `abs`, `_itoa`, `_i64toa`, `labs`, `_ltoa`, `strtol`, `strtoul`, `__toascii`, `_tolower`, `toupper`, `_toupper`, and `_ultoa`.

Versions of `atoi()` and `strtol()` are found in `utilities.lib`; refer to `utilities.h` for more information.

### Searching and sorting functions

The library functions `bsearch`, `_lfind`, `_lsearch`, and `qsort` should work in the RIM Wireless Handheld environment. Testing for these functions has not been completed at this time.

## Incompatible functions

These function groups are not compatible, or contain functions that are not compatible with the RIM Wireless Handheld.

### Byte classification (multibyte) functions

The RIM Wireless Handheld does not support multibyte characters.

### Character classification functions

Because the RIM Wireless Handheld does not support multibyte or wide characters, multibyte or wide character functions are also not supported. Many of the other `isxxx()` functions and macros are locale-dependent. Different locales have different sets of uppercase and lowercase characters.

### Debug functions

The RIM Wireless Handheld does not support the compiler library debugging support functions. Other functions are provided to support the debugging of wireless handheld applications. For example, short debugging messages can be output to the debug output window when running Microsoft Developer Studio by calling `RimDebugPrintf`.

### Directory control functions

The RIM Wireless Handheld file system is different from those used on desktop systems. As a result, the directory control functions are not supported by the RIM Wireless Handheld.

### Error and exception handling functions

The RIM Wireless Handheld environment does not support exception handling. Serious failures, such as page faults, are handled by the system function `RimCatastrophicFailure`, which can be called by application code when an unrecoverable error is detected.

### File handling functions

The RIM Wireless Handheld file system is different from those used on desktop systems. As a result, the file handling functions are not supported by the RIM Wireless Handheld.

### Floating-point functions

Because the RIM Wireless Handheld has no floating point co-processor and does not support application handling of traps and exceptions, floating point functions cannot be used, unless all floating points are implemented by emulation without calls to the operating system.

The following functions are incorrectly classified as floating point functions, and can be used: `div`, `labs`, `ldiv`, `_lrotl`, `_lrotr`, `_max`, `_min`, `rand`, `_rotl`, and `_rotr`.

### Input and output functions

Any stream, file, or console input/output functions are not compatible with the RIM Wireless Handheld environment. The RIM Wireless Handheld environment provides different mechanisms for keyboard input, LCD output, and file system access. Because of how they are implemented in the compiler library, this category also includes `sprintf` and `scanf`. These cannot be used in a RIM Wireless Handheld application. Use `RimSprintf` or `RimVSprintf` instead of `sprintf`.

The port I/O functions, such as `inp` and `outp`, do not cause an error during compilation, linking, or loading; however, because RIM Wireless Handheld applications run in a protected environment, calling these functions cause a protection fault at run time.

### Locale-dependent functions

The RIM Wireless Handheld environment does not support locales, multibyte characters, or wide characters; however, it is useful to have many of the locale dependent functions compatible. These functions were in the C library before the `setlocale` function.

### Memory allocation functions

The standard C memory allocation functions cannot be used in the RIM Wireless Handheld environment. Use `RimMalloc`, `RimRealloc`, and `RimFree` instead.

The global C++ operators `new` and `delete` can be used. They are translated into calls to `RimMalloc` and `RimFree` respectively. The `new` operator differs from the standard one in that, if there is insufficient memory to perform the allocation, it returns a `NULL` pointer instead of throwing an exception.

### Process and environment control functions

Because the process model of the RIM Wireless Handheld environment is quite different from that of desktop systems, the compiler C library process and environment control functions are not applicable.

The functions `setjmp` and `longjmp` are also in this class. They are not available because the library implementations make system calls to perform stack unwinding. You could write a version of these functions that would work on the RIM Wireless Handheld, if it is acceptable not to call the destructors of C++ stack objects between the `setjmp` and the `longjmp`.

### String manipulation functions

Many of the string manipulation functions are locale dependent or require the use of `malloc`. These functions are not supported by the RIM Wireless Handheld environment.

Some string manipulation functions can be found in the `utilities.lib` library; refer to `utilities.h` for a list. For more information, refer to String Utilities in the System Utilities API.

### **System call functions**

Windows system call functions cannot be used in the RIM Wireless Handheld environment. Because of how the handheld operating system allocates application stacks, Windows functions are not safe to call even when they are running in the simulator.

### **Time functions**

The RIM Wireless Handheld operating system represents time differently from desktop systems. As a result, time functions are not compatible with the handheld environment.

# Index

## A

- and, 64
- API, 116
  - RadioRegister(...), 79
- application server, 69
- applications
  - loading, 30
  - restarting, 49
- asynchronous send See inter-process communications
- attachments, 88

## B

- background application
  - bringing to foreground, 69
- backlighting, 37
- bitmaps
  - converting existing, 117
  - creating, 116
  - definition file, 117
  - displaying, 117
  - maximum size, 116
- bitmaps.exe, 117
- blocking send See inter-process communications

## C

- choice boxes, definition, 85
- class hierarchy, 98
- classes
  - BitField, 99
  - database, 98
  - DatabaseRecord, 99
  - DatabaseView, 99
  - DataBuffer, 99
  - DbrecordBitField, 99
  - FlashObject, 98
  - PersistentStore, 99
- classes overview, 50
- code tutorial
  - add attachment data, 91
  - attachment type, 90
  - messaging, 89
- Compiler, 72

- compiler optimizations, 72
- contacts
  - adding email, 102, 110
  - associating fields with records, 106
  - defining views, 105
  - displaying, 103
  - displaying different list views, 107
  - editing, 106, 109
  - removing a field, 112
  - removing email, 102
  - saving, 111
  - sorting, 103
  - storing, 103
  - switching list views, 108
  - updating lists, 108
- controls, 86
- converting existing bitmaps, 117
- creating bitmaps, 116
- creating fonts, 118

## D

- database
  - getting first and last name, 101
  - setting first and last name, 101
  - setting up, 101
- database / file system
  - reading, 70
  - record pointer table, 72, 73
  - referencing data, 72
  - using the PointTable edition counter, 73
- DbPointTable(), 72
- debugging
  - functions (compatibility), 128
  - using simulator, 31
- definitions
  - choice boxes, 85
  - dialog boxes, 86
  - edit boxes, 84
  - fields, 84
  - lists, 84
  - menus, 83
  - screens, 82
  - separators, 85

## Index

- status boxes, 85
- trackwheel, 82
- design guidelines
  - controls, 86
  - keyboard, 86
  - trackwheel, 86
- development environment
  - configuring, 11
- device errors
  - 95, 49
  - 96, 49
- dialog boxes
  - definition, 86
- dll, 12
- DMP files, 42

### E

- edit fields
  - definition, 84
- events
  - MESSAGE\_RECEIVED, 80
  - MESSAGE\_SENT, 79
- events, notification, 79
- example programs, 66
- exception handling, 128

### F

- fields, 83–85
  - choice boxes, 85
  - definitions, 84
  - edit boxes, 84
  - lists, 84
  - separators, 85
- file
  - streamed, 71
- file system
  - reading, 70
  - services, 70
  - terms, 70
- files
  - \*.DLL, 35
  - \*.DMP, 42
  - debug.dat, 20
  - flash simulation, 42
  - sdkradio.dll, 35
- fixed-width font, 118
- flash memory, 50, 70
  - reading from flash, 50
- flash memory allocation log, 42
- flash memory file system, 42
- flash simulation files, 42
- fonts
  - creating, 118

- creating the resource DLL, 121
- default fonts, 118
- naming, 119
- testing the resource DLL, 122

- foreground application
  - sending to background, 69
- functions
  - \_lrotl, 128
  - \_lrotr, 128
  - \_lsearch, 127
  - \_max, 128
  - \_min, 128
  - \_rotl, 128
  - \_rotr, 128
  - bsearch, 127
  - div, 128
  - inp, 128
  - ldiv, 128
  - outp, 128
  - qsort, 127
  - rand, 128
  - RimFree, 129
  - RimMalloc, 129
  - RimRealloc, 129
  - RimSprintf, 128
  - RimVSprintf, 128
  - setlocale, 129
  - sprintf, 128
  - sscanf, 128

### G

- global constructors, 56
- graphics services, 115

### H

- handheld
  - activating applications, 52
  - LCD, 52

### I

- icons, 118
  - creating, 52
- installing the SDK, 10
- interface, user, 52
- inter-process communications
  - asynchronous (non-blocking) send, 68
  - synchronous (blocking) send, 69
- invoking a menu, 83

### K

- keyboard, 86
- keypad, 69

**L**

- launching the simulator, 31
- lcd size, 37
- lcdfonts.exe, 120
- lists
  - defining views, 105
  - definition, 84
  - displaying different views, 107
  - sorting, 103
  - switching views, 108
  - updating, 108
- log structured file system, 70

**M**

- macros
  - argument access, 127
  - locale dependent, 127
  - RIM\_DEFINE\_DEPENDENCY, 65
  - RIM\_DEFINE\_VERSION, 65
- memory
  - flash, 50
  - RAM, 50
  - structure, 100
  - use, 50
- memory mapped file access, 72
- menu items
  - cancel menu item, 83
  - definition, 83
  - design, 83
  - designing, 83
  - hide menu, 83
  - invoking a menu, 83
- message passing, 68
  - asynchronous send, 68
  - synchronous send, 69
- MESSAGE\_RECEIVED event, 80
- MESSAGE\_SENT event, 79
- messages
  - structure, 88
  - supported types, 88
- Microsoft Developer Studio, 11
- modem
  - radio simulation control panel dialog box, 37
  - simulating using a physical modem, 43
  - simulating using the file system, 44
  - simulation with rap modem, 43
  - using the pager as a modem, 44
- MPAK
  - defined, 78
- multiple applications, 116
- multitasking, 8

**N**

- network, 79, 80
  - destination out of coverage, 79
- non-blocking send See inter-process communications

**O**

- opening
  - simulator, 31
- operating system services
  - message passing, 68
  - multiple applications, 69
- options
  - overview, 95
- orphan records, 71
- OsLoader.exe, 30

**P**

- packet transmission, 40
- packets
  - receiving, 80
  - transmitting, 79-??
- PointTable edition counter, 73
- processor, 8
- program loader
  - alloc command, 18
  - batch command, 18
  - command line options, 17
  - dir command, 19
  - erase command, 19
  - help command, 20
  - load command, 20
  - troubleshooting, 24
  - ver command, 23
  - wipe command, 23
- proportional font, 118, 119

**R**

- radio
  - simulation control panel dialog box, 37
- radio simulation control panel dialog box
  - packet transmission section, 40
- RadioGetMpak(...), 80
- RadioGetSdu(...), 80
- RadioResumeReception(...), 80
- RadioStopReception(...), 80
- RAM, 50
- random access to files, 71
- reading from the file system, 70
- receiving packets, 80
- record handle representation, 72
- record pointer table, 72

## Index

- address, 72
- references
  - RIM documentation, 5
  - RIM web site, 5
- registers, displaying contents of, 45
- resource DLLs, 120
  - creating, 120
  - creating fonts, 121
  - creating tunes, 122
  - testing the font DLL, 122
  - testing tunes, 124
- restarting applications, 49
- RFC 822, 89
- ribbon
  - messages, 94
  - overview, 94
  - using, 94
- ribbon icons See icons, creating, 52
- RIM co-operative scheduler, 48
- RimCatastrophicAPIFailure(), 21
- running the simulator, 30

## S

- screens
  - definition, 82
- SDK
  - multi-tasking, 8
  - processor, 8
- sdkradio.dll, 35
- SDU
  - defined, 78
- search, 82
- sending
  - data packets, 79
- separators
  - definition, 85
- sequence numbering of packets, 79
- serial input/output, 40
- serial port, 33
- simulating
  - battery environment, 37
  - holster environment, 37
  - serial input/output, 40
- simulation menu
  - battery, 37
  - holster, 37

- simulator
  - applications for full simulation, 35
  - backlighting, 37
  - command line options for rap modem, 43
  - command line switches, 33
  - data pointer errors, 73
  - ESCAPE key, 36
  - integrating with Visual Studio, 31
  - launching, 31
  - LCD, 37
  - opening, 31
  - registers, displaying contents of, 45
  - running, 30
  - simulating using a physical modem, 43
  - using, 35
- simulator.exe, 30
- stack size, 57
- standard C library
  - compatible functions, 127
- status boxes
  - definition, 85
- streamed file, 71
- style guidelines, 86
- symbols, 118
- synchronous send See inter-process communications

## T

- tasks
  - application development, 49
  - yielding, 49, 69
- threads, 49, 69
- trackwheel, 69, 82, 86
- transmitting packets, 79-??
- tunes
  - creating the resource DLL, 122
  - testing the resource DLL, 124

## U

- UI Engine
  - compatibility with, 100
  - interaction with, 109
- user interface, 52

## W

- watchpuppy, 49





© 2002 Research In Motion Limited  
Published in Canada