RIM 902M Radio Modem

Software Development Kit

Developer Guide

Version 2.5

RIM 902M Radio Modem SDK – Developer Guide, Version 2.5
Last revised: 7 January 2002

Part Number: MAT-02445-002

Research In Motion Limited
295 Phillip Street
Waterloo, Ontario
Canada N2L 3W8
Tel. (519) 888-7465
Fax (519) 883-4940
Web site: www.rim.net/oem
E-mail: info@rim.net

Printed In Canada

Contents

1 Introduction

This guide explains how to use the RIM 902M Radio Modem Software Development Kit (SDK) to develop on-board radio modem applications.

About this guide

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

This guide explains these topics:

- installing and configuring the SDK and development environment
- loading applications onto a physical radio modem
- testing applications using the RIM 902M Radio Modem Simulator
- operating system, file system, and radio communications
- reference information on radio modem APIs

Other documentation

RIM provides the following additional documentation:

- *Integrator Guide*

  The *Integrator Guide* explains how to integrate the RIM 902M Radio Modem into devices, including mechanical integration, power characteristics, modem interfaces, and antenna selection and placement. The *Integrator Guide* can help you decide whether to use this SDK to develop an on-board application or whether to use an external processor.

- *Database API Reference*

  The *Database API Reference* explains Radio Modem Database API functions, which provide optional functionality in addition to the File System API, which is described in this *Developer Guide*.

- *Radio Modem Simulator Online Help*

  The Help provides information for using the simulator, including procedures, simulator options, and troubleshooting. On the Simulator **Help** menu, click **RIM Simulator Help**.

- *Radio Access Protocol (RAP) Programmer Guide*

  If you decide to use an external processor rather than developing an on-board application, the *RAP Programmer's Guide* explains how to program wireless communication for your applications using RAP, a RIM proprietary communications protocol.

- README.txt

  The `README.txt` file is installed with the SDK. It provides information on system requirements, new features, and known issues, as well as any last-minute documentation updates.

Technical support

If you need help getting started, or if you have any questions about the radio technology or its integration into your platform, contact RIM:

Email:	oemsupport@rim.net
Phone:	+1 (519) 888-7465 extension 5200
Fax:	+1 (519) 885-3110
Web:	http://www.rim.net/oem

About the RIM Radio Modem SDK

This SDK includes the tools you need to develop applications for the RIM 902M Radio Modem Simulator.

The SDK provides a powerful development environment that uses Microsoft Visual Studio 6.0 or later (Visual C++ 6.0 or later) on Windows 95/98/NT/2000.

The SDK package includes the following items:

- RIM 902M Radio Modem APIs
- reference documentation for all APIs
- source code examples
- application utilities
- PC-based radio modem simulator

Development environment

The RIM 902M Radio Modem SDK uses the libraries and features in Microsoft Visual Studio 6.0 or later. You must buy and install Microsoft Visual Studio separately from this SDK.

You can use all the development facilities of Visual Studio, including the integrated development environment (IDE) and debugging tools.

Radio modem 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 is then ported to the radio modem operating system. Applications must not make calls to the Windows API, even when running in the radio modem simulator.

Simulator

The simulator included with the SDK enables you to test applications on the radio modem operating system without having to load the .dll files onto an actual device. The simulator supports all radio modem functionality.

The SDK includes a Mobitex network simulator that enables you to test the network communication of an application without connecting to a live Mobitex network.

In addition, you can connect the simulator to a modem through the computer serial port to enable simulated applications to communicate over the live Mobitex network.

Radio modem APIs

The SDK provides the following APIs to enable applications to use the radio modem’s operating system.

Radio API

The Radio API provides access to the network using simple API function calls to send and receive data. You do not need extensive knowledge of the Mobitex network to use these function calls.

Radio events provide information on the status of incoming and outgoing packet communications. Radio events are announced to applications through the message system.

Serial Communications API

The Serial Communications API provides access to external serial ports on the radio modem. Low-level hardware functionality is built into the operating system; you do not need to program complex serial functions.

The Serial Communications API enables you to configure serial port parameters and use data set ready (DSR) and data terminal ready (DTR) signalling. The RIM 902M Radio Modem also has the capability for hardware flow control with OS support.

File System API

The File System API provides access to the radio modem’s persistent memory. To make more efficient use of the flash memory, the radio modem manages files in a unique manner. You can reference records in the database directly, without copying the data into RAM first.

If your application requires additional database functionality, you can use the Database API. Refer to *Database API Reference*.

System API

The System API provides a wide range of functionality to radio modem applications, including thread management, message handling, and task switching. Other system-level functionality includes memory allocation, timers, and access to the system clock.

LED API

The LED functions enable you to control, for testing purposes, the LEDs on the radio modem’s test board related to wireless network coverage and messages.

2
Getting Started

This chapter explains how to perform the following tasks:

- install the SDK
- configure Microsoft Visual Studio
- integrate the RIM 902M Radio Modem Simulator for debugging

Installing the SDK

You must install Microsoft Visual C++ 6.0 before you install the RIM Radio Modem SDK.

1. Double-click the **setup.exe** file for the RIM 902M Radio Modem SDK. The InstallShield Wizard Welcome screen appears.
2. Click **Next**. The License Agreement for the Radio Modem SDK appears.
3. Review the license and, if you agree to its terms, select **I accept 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 use of the SDK to your own user account or to allow all users to access the SDK.
7. Click **Next**. The Destination Folder screen appears.
8. Accept the default folder, or click **Browse** and select another folder.
9. Click **Install**. The InstallShield Wizard installs the SDK.
10. Click **Finish** to complete the installation process.

Setting up projects

The installation process for the RIM Radio Modem SDK creates a project type in Microsoft Visual Studio for radio modem applications. This project type ensures that the project has the appropriate settings within the Visual Studio development environment.

Project settings

The RIM OEM SDK application project type has the following settings.

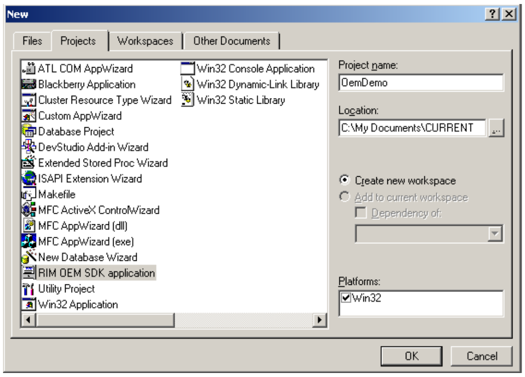
On the **C/C++** tab:

- In the **Preprocessor** category, the `include\application` and `include\application\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.

On the **Link** tab, in the **General** category, the files `OsEntry.obj`, `RimOs.lib`, and `libc.lib` are included in the **Object/library modules** field.

To create a new project

1. Start Visual C++ 6.0.
2. On the **File** menu, click **New**. The **New** screen appears.
3. Click the **Projects** tab.



Project name: OemDemo

Location: C:\My Documents\CURRENT

[✓ Create new workspace]

Platforms: [✓ Win32]

New Projects screen

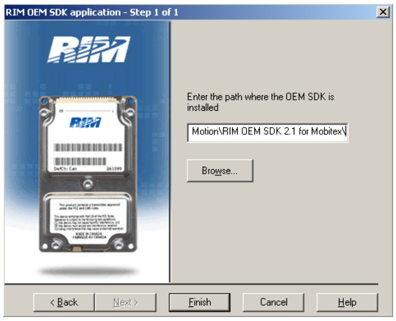
4 From the projects list, select **RIM OEM SDK application**.

5 In the **Project name** text box, type a project name.

6 In the **Location** text box, select the location in which the project should be installed.

7 Click **OK**.

A setup screen appears.



Enter the path where the OEM SDK is installed

Motion\RIM OEM SDK 2.1 for Mobitex\

Installation setup screen

8 If you have installed the RIM OEM SDKs for both Mobitex and DataTAC, select which SDK to use for this application.

9 Select the folder in which the RIM OEM SDK is installed.

10 Click **Finish**.

The New Project Information screen appears, with a summary of new project information.

11 Verify that the information is correct and click **OK**.

The new project appears in the Microsoft Visual Studio project workspace.

If you go to the **File View** in Microsoft Visual Studio and look at the source file for the project, the App Wizard has created a template for a typical radio modem application.

Refer to “Writing a radio modem application” on page 15 for more information.

Setting debug properties

When you create a new project for a radio modem application, you must set the debugging options.

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\
   RIM 902M OEM Software Developer's Kit 2.5\
   simulator\OSLoader.exe
   ```
6. In the **Program arguments** field, type **`OsOemMb.dll`**.

   You can also type other command line options (refer to “Simulator command line options” on page 22).
7. In the **Working directory** field, specify the folder in which your application .dll files are located.
8. Click **OK**.

   When you debug this project in Visual C++ 6.0, it runs the application in the RIM Radio Modem Simulator.

   Refer to “Using the simulator” on page 25 for more information on using the simulator.

3
Writing applications

This chapter explains the basic structure of a typical radio modem application.

Writing a radio modem application

This section briefly outlines the required components of an application developed for the RIM 902M Radio Modem.

Entry point

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

```
void PagerMain( void )
```

Registering the application

RIM 902M Radio Modem applications must have two variables defined globally: the version string and the stack size.

The version string is used to register the application with the task switcher in the operating system. The name appears in the debugger only.

```
char VersionPtr[] = “My Application”;
```

The system uses the stack size when creating the initial thread for the application. The value should be sized according to the needs of your application.

```
int AppStackSize = 2048;
```

Entering the message loop

Radio modem applications spend a significant amount time in the message loop. This code block should retrieve application messages and perform required operations. It is not necessary for all applications to respond to all messages.

The typical structure for this loop is a non-terminating loop with a switch statement to delegate message processing to different functions. This is illustrated in the following code.

```
#include “RimOEM.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;
    }
  }
}
```

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

The `RimGetMessage` function must be called explicitly by an application to retrieve pending application messages. If `RimGetMessage` is not called, all other applications remain blocked and the radio modem becomes unresponsive.

Example programs

Source code for the following sample applications is available in the `samples` folder in the SDK installation folder:

Project	Description
Database.dsp	This application provides a basic example of how to use the database DLL.
GPS.dsp	This application interacts with a Global Positioning System (GPS) device using the TAIP protocol.
I2C.dsp	This application demonstrates how to use the I^2C driver (refer to “I^2C Driver” on page 168 for more information).
SerialDemo.dsp	This application demonstrates how to use the serial port.

4
Using the simulator

The simulator included with the SDK enables you to test applications on the radio modem operating system without having to load the .dll files onto an actual device.

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 Mobitex 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.

Running the simulator with Microsoft Visual Studio

As explained in “Setting debug properties” on page 14, 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 Radio Modem Simulator environment during debugging.

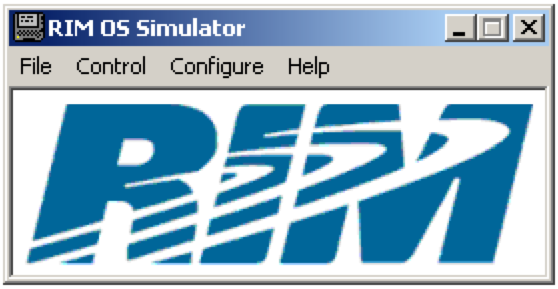
When you debug a project in Visual C++ 6.0, the simulator automatically starts. You can then run the application in the simulator while using all of the debugging facilities, such as breakpoints, in Visual C++ 6.0.

Running the simulator in Windows

1 On the **Start** menu, select **Programs > Research in Motion > RIM 902M OEM Software Developer's Kit 2.5 > Simulator**.

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

The RIM OS Simulator window appears.



RIM Radio Modem simulator

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.** Select the appropriate OS .dll file in the `simulator` folder, such as `OsOemMb.dll`.

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.

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 the **Browse** button.

6 In the **Select Applications** window, select the application .dll files to load and click **Open**. You can select several applications.

7 After you have finished selecting applications, click **Cancel**.

8 On the **Control** menu, click **Start Simulation**.

Refer to “Using the simulator” on page 25 for more information.

Running 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] OsOemMb.dll [DLLs]
```

For example:

```
OSLOADER.EXE OsOemMb.dll SampleApp.dll
```

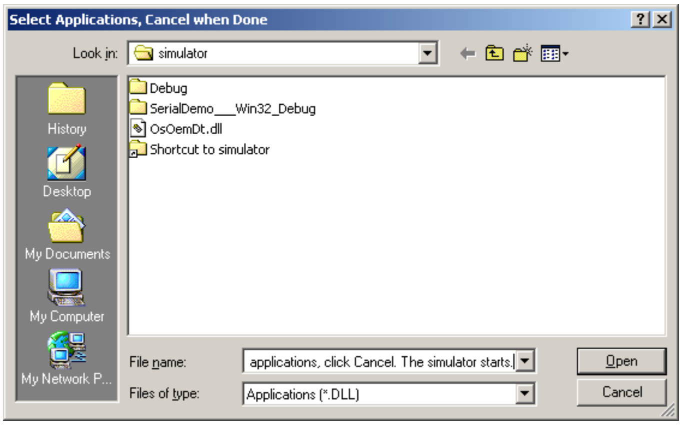
where `OsOemMb.dll` specifies the operating system to use and `SampleApp.dll` is the application to load into the simulator.

To specify application .dll files at the command prompt, they must be included in the PATH environment variable.

Refer to "Simulator command line options" on page 22 for more information on options.

3 If you did not specify any .dll files at the command prompt, a Select Application dialog box appears.

Select each .dll file that you want to load and click **Open**. After you have finished selecting applications, click **Cancel**.



File name: applications, click Cancel. The simulator starts.
Files of type: Applications (*.DLL)

Select Applications window

The simulator starts.

Refer to "Using the simulator" on page 25 for more information.

Simulator command line options

The simulator environment can be configured using the following command line options.

Option	Description	Example

`/Sx`	`/Sx` specifies the Windows serial port that is to be used instead of the RIM 902M Radio Modem's physical serial port. Omitting this argument means that no serial port is used, and applications cannot open a serial port. Entering two arguments is valid since the RIM 902M Radio Modem has two serial ports. The first port specified is the MASC port, and the second specified is the other port.	`/S1`
		`/S2`
		`/S3`
		`/S4`

`/Rx`	`/Rx` specifies which serial port the simulator uses for a RAP modem. The RAP modem is used instead of the physical hardware in the simulation environment. If this argument is omitted, no serial port is used, and simulated radio communications are not possible.	`/R1`
		`/R2`
		`/R3`
		`/R4`

`/Pf` `/Ps`	`/Pf` forces prompting for more applications to be loaded, while `/Ps` suppresses prompting for more applications. By default, the simulator prompts for applications only if no applications are specified on the command line.	`/Pf`
`/Fx`	`/Fx` specifies the amount of simulated flash memory in kilobytes. If this argument is omitted, the default size of 2048 KB is used.	`/F2048`
`/L`	`/L+` enables simulation of flash memory timing. `/L-` disables simulation of flash memory timing.	`/L+`
`/E`	`/E` specifies that the simulator should erase the flash allocation log completely. Flash memory allocation information no longer exists and a default allocation is used until a new allocation is specified. This option must be specified when all flash memory allocation log entries have been used.	`/E`

Option	Description	Example
`/Dx`	`/Dx` specifies the amount of simulated flash memory (in 64-KB sectors) to be used for file system data storage. If this option is omitted, the previous amount is preserved, unless `/E` is also specified. If `/E` is specified, a default amount is used.	`/D5`
`/Ax`	`/Ax` specifies the amount of simulated flash memory (in 64 KB sectors) to be used for OS and application code storage. If this option is omitted, the previous amount is preserved, unless `/E` is also specified. If `/E` is specified, a default amount is used.	`/A6`
`/M<n>`	`/M` sets the debug level for memory checking, where *n* is 0 to 4 (0 is lowest level, 4 is highest). `/M0` - use the main heap only `/M1` - use the private heap and main heap `/M2` - use both heaps, with bounds checking `/M2` - use both heaps, with free pointer checking `/M4` - use both heaps, with free pointer checking and bounds checking The default is `/M4`.	
`/N`	/N deletes the flash memory file system before initialization and creates a new flash memory file system, instead of resuming from the last simulation. /N is disabled by default	

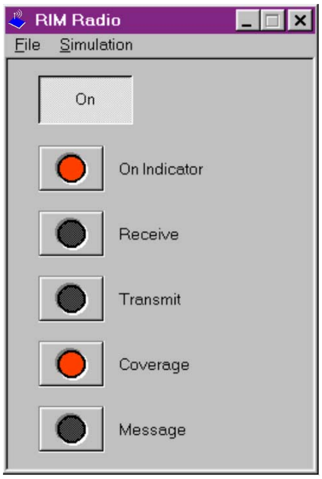
Option	Description	Example
`/T [Flags]`	`/T` controls stale pointer trapping. By default, the simulator moves the location of the simulated file system at various intervals. The simulator then marks the old location as invalid memory. When pointers to the old file system location are used, page faults occur. This indicates that the pointers should have been reloaded from the handle table. The `/T` option can be followed by these symbols: + Turn subsequent options on (default) - Turn subsequent options off W Trap on flash memory write Y Trap on yield G Trap aggressively (simulator moves flash memory five times as often)	`/TW-Y`
`/RSIM=####`	This option sets the address (MAN number) to simulate, when you are simulating network communication using the file system.	`/RSIM=12345`

`/RDIR=DIR`	The `/RDIR` option sets the directory to use for the simulation, when you simulate network communication using the file system. The default is the current directory.
	Radio simulators must point to the same directory to communicate with each other.
		`/RDIR=`
		`c:\network`

`Filename`	Any argument that does not begin with a slash is assumed to be an application .dll file name. You can specify multiple .dll files at the command line. File names containing spaces must be surrounded by quotes. If you do not specify the full path to the file, the file must be located in the working directory.	`app2.dll`

Using the simulator

When you start an application in the simulator, a window appears that represents the RIM 902M Radio Modem test board.



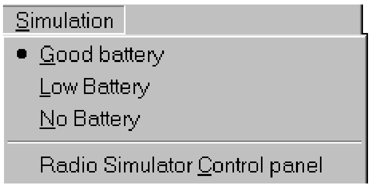
RIM Radio Modem simulator

You can click the **On** button to simulate turning on or off the radio modem. The **On Indicator** LED shows whether the modem is on.

The **Receive** and **Transmit** LEDs indicate when the application is receiving or sending data. The **Coverage** LED indicates whether or not the application is simulated as within wireless network coverage (refer to “Setting radio simulation options” on page 29 for more information). The **Message** LED indicates when a message has been received.

Simulating battery conditions

The radio modem simulator enables you to simulate various battery conditions of the radio modem, for applications in which the RIM 902M Radio Modem is battery-powered. On the **Simulation** menu, select a battery condition: **Good battery**, **Low battery**, **No battery**.



Simulation menu

Simulating serial I/O

The simulator can use one or two computer serial ports to simulate the RIM 902M Radio Modem’s serial ports. You must specify the `/Sx` command line option when you start the simulator (refer to “Simulator command line options” on page 22). If no `/Sx` option is specified or the serial port is not available at startup, you cannot open or use the serial ports from applications running on the simulator.

If you want to simulate the RIM 902M Radio Modem 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 a host-based RIM 902M Radio Modem application 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 flash memory

To simulate the non-volatile nature of flash memory, the RIM 902M Radio Modem simulator loads the flash memory contents from a file on startup and saves them back to the file on exit. The state of the simulated flash memory is preserved in a file named `FILESYS.DMP` in the working directory.

When the 902M Radio Modem simulator starts, it determines the simulated flash memory size. The size can be specified using the `/F` command line option. The default size is 2048 KB.

The simulator checks to see if there is a file named `FILESYS.DMP` in the working directory. If the file exists, its size must be equal to the simulated flash memory size; otherwise the simulator reports an error and does not start. If the file does not exist, the simulator creates it with the simulated flash memory size. The simulator always saves the simulated flash memory contents to `FILESYS.DMP`.

You must design your applications in a way that minimizes the performance impact of the flash memory file system. Refer to "Understanding the file system" on page 68 for more information.

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

The allocation of these areas is performed by writing an entry to the flash memory allocation log, which is a special data structure stored within the fixed-use area. You can erase this log using the `/E` command line option.

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 simulator does not allow any more entries to be written until the log is erased.

If the simulator finds no valid log entry in the `FILESYS.DMP`, a default flash memory allocation is used.

Using a physical modem

You can use the simulator with an actual RIM 902M Radio Modem. The simulator manages network traffic by communicating with a physical modem using a live Mobitex network. The modem must have a valid subscription, and you must be operating in an area that has Mobitex wireless network coverage.

Set the `/R` command line option to specify which radio serial port to use (refer to "Simulator command line options" on page 22).

Applications use the specified serial port to send a Mobitex packets (MPAKs) to the physical modem. MPAKs and status information received by the physical modem are sent to the simulator running on the computer, through the serial port, and are passed on to the applications.

When using a physical modem with the simulator, you can only change wireless network coverage conditions by varying the placement and position of the antenna. The dialog box for manually controlling wireless network coverage situations is unavailable when running the simulator with a physical modem, because network conditions are determined by what the actual modem experiences on the live network.

Simulating network communication

The SDK includes a Mobitex network simulator that enables you to test the network communication of an application without connecting to a physical modem and a live Mobitex network.

Network simulation has several advantages:

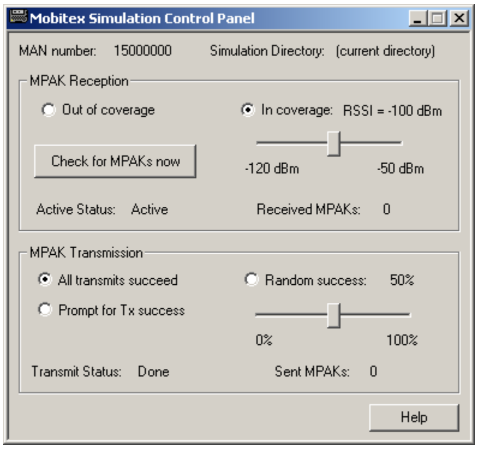
- You do not need any external hardware.
- You do not need a Mobitex account.
- You can simulate varying coverage situations.
- You can monitor what is sent, without extra tools.

You can test code running on the simulator against a server by writing a simple host-side application that uses the file system to simulate network traffic. Refer to “Writing a host-side simulation” on page 37 for an example.

Setting radio simulation options

A separate control panel enables you to set coverage conditions to simulate Mobitex network conditions.

On the **Simulation** menu, click **Radio Simulator Control Panel**.



MAN number: 15000000 Simulation Directory: (current directory)

[ ] Out of coverage [✓ In coverage:] RSSI = -100 dBm

Active Status: Active Received MPAKs: 0

[✓ All transmits succeed] [ ] Random success: 50%

[ ] Prompt for Tx success

Transmit Status: Done Sent MPAKs: 0

Radio simulation control panel

MAN number

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

Simulation Directory

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

MPAK Reception

The MPAK Reception section enables you to simulate modem coverage conditions and shows information on received MPAKs.

Out of coverage

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

In coverage

When you select **In coverage**, you can use the slide bar to control the received signal strength indicator (RSSI) 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 you are likely to see on the actual network.

Many Mobitex modems also measure the signal strength in dB microvolts (dBμV). 0 dBμV is the same as -113 dBm. Therefore, -7dBμV equals -120 dBm and 63 dBμV equals -50 dBm. Many applications also report coverage in percent. The percent scale is arbitrarily assigned by the applications, and no universal mapping exists from percentage to RSSI.

Check for MPAKs now

Click the **Check for** MPAK**s now** button for the simulator to look for MPAKs immediately. This process is useful in two situations:

- to increase the frequency with which the simulator checks for MPAKs; by default, the simulator checks the hard drive for MPAKs once every 10 seconds to simulate the delays that can be introduced in sending unsolicited MPAKs to a particular modem
- to increase the speed of simulated retries, so that MPAKs 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 (`RadioStopReception` has been called).
Active	The modem is in network coverage on the network.
Checking...	The radio is checking for MPAKs on 10-second intervals, or the **Check for** MPAK**s now** button was pressed.

State	Description
Checking for 10 sec…	The radio is checking continuously due to a MPAK recently sent or received.
Out of coverage	The modem is out of network coverage.

Received MPAKs

The **Received** MPAK**s** field displays the number of MPAKs that have been received from the simulated network.

MPAK Transmission

The MPAK Transmission section enables you to control the simulated transmission of MPAKs and shows information on of sent MPAKs.

All transmits succeed

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

Prompt for Tx success

When you select this option, every time an MPAK submitted for transmission, a dialog box appears prompting the user to select whether or not the MPAK will be transmitted successfully. The MPAK 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, an MPAK can be pending in the modem for tens of seconds.

Random success

Using the slide bar, you can select the probability of a particular MPAK being sent successfully to the network. The delay for successful MPAKs is about 1.5 seconds, while the delay for unsuccessful MPAKs 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 MPAK that was submitted for transmission. This MPAK can be in one of the following states:

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

How the file system simulates the network

A directory on your computer represents the network. By default, the current directory is used. You can specify a directory using the `/RDIR` command line option.

Each MPAK in the simulated network is represented as a file in this directory. When the simulator sends an MPAK, a file is created; when the modem receives the MPAK, the file is deleted. The simulator checks for the presence of files whose names indicate that they contain MPAKs addressed to the simulator.

Each file name includes an 8-digit MAN number, which indicates that the file contains an MPAK that is routed to this number.

The binary representation of the file equals exactly the contents of the entire raw MPAK. For example, a 12-byte STATUS MPAK produces a file 12 bytes in length. The MPAKs are formatted the same way that they are represented on the Mobitex network, including the MPAK header, as described in “Data packet structure” on page 51.

Modem startup file

When the simulator starts, the simulator creates a file named `[MAN].MAN.` For example, a modem with the MAN number 12345 would create the file 00012345.MAN. By default, the simulator uses the MAN number 15000000. You can specify a different MAN number using the `/RSIM` command line option.

The modem startup file indicates that a modem simulation by that MAN is running, and that traffic can be sent to it. During simulation, this file remains open to prevent any other simulation with the same MAN from overwriting the file. When the simulator is stopped, other simulations can write to the file.

If you write a separate host-side simulation, and you check for the existence of a simulator, you must also check for the file using a directory function, such as `t_findfirst`.

When trying to transmit to a modem that does not have a modem startup file, the radio modem simulator returns the MPAK with a status of `NO_TRANSFER`. A simulation that is not currently running is treated as an invalid subscription, and cannot be addressed by another simulator.

Limitations of Mobitex network simulation

This section describes some limitations of Mobitex network simulation.

Because the file system has no built-in Mobitex intelligence, the radio modem simulator performs some of the network simulation. In general, the recipient's simulator performs all processing on an MPAK after it is sent to the network. The only exception is the fast returning of MPAKs sent to MAN numbers that do not exist. (There is no simulator on the other side to return the MPAKs.)

The responsibility for the processing of MPAKs is as follows:

Sender-side processing:

- Set `TIME` field in the MPAK
- Return MPAK to sender if the recipient does not exist
- Return MPAK with a status of `CONGEST` if all 10 sequence numbers for the destination MAN number are already used

Receiver-side processing:

- Return `POSACK` MPAKs
- Return Mailbox MPAKs and simulate Mailbox behavior
- Simulate coverage presence or absence
- Simulate RSSI

Return MPAKs while out of coverage (both slow and fast returns)

Network features that are simulated

The following network features are simulated.

Network feature	Description
POSACK	The simulator is able to simulate Mobitex POSACK behavior. With POSACK, a copy of the MPAK sent is delivered to the sender once the packet has been delivered to the recipient.
MAILBOX	The simulator is able to simulate mailboxing of MPAKs, provided that a simulator running the MAN number is active and not in coverage. The receiver's simulator deals with mailbox handling. The actual network will expire a mailboxed MPAK after a period of typically 8 hours. The simulator, however, does not expire mailboxed MPAKs.
MPAK TIME field	The Mobitex network timestamps each MPAK as it is received. The sending radio simulator also performs this function. As such, when processing MPAKs outside of the simulator, the time information remains set to what it was in the MPAK contained in the file. Timestamps assigned to files by the operating system are not used.
MPAK sent to unsubscribed MANs returned	Whenever an MPAK is sent to a particular MAN, the simulator checks to see if a file exists to indicate the presence of a simulator for that MAN. If one does not exist, the simulator instead sends the MPAK to itself with a traffic state of `NO TRANSFER`. The receiving simulator handles all other returns of MPAKs. If no receiving simulator is running there is no code present to actually return the MPAK.
MPAK sent to out-of-coverage modems returned slowly	When an MPAK is sent to a modem that is out of coverage, the network typically tries to contact that modem over a period of at least 40 seconds. If all transmit attempts fail, the MPAK is returned to the sender. If a number of attempts have been made at sending an MPAK to a modem that is out of coverage, the network eventually stops attempting to send to that modem until the modem indicates that it is back in coverage. As such, MPAKs are returned as not delivered in a matter of seconds. The simulator spreads its simulated retries over the course of 30 seconds. Clicking the **Check for MPAKs now** button can accelerate this process. The receiving simulator returns the MPAKs, so the receiving simulator must be running.

Network feature	Description
MPAK receive delays	On the actual network, depending on the settings, the radio modem only communicates with the network occasionally, approximately every 40 seconds. Thus, any MPAKs sent to a modem may be delayed by up to that period, as the base station must hold the MPAKs until it knows that the radio modem can accept its signals. Checking for MPAKs from the file system only every 10 seconds simulates this delay. This is a shorter delay than on the actual network, but it is still noticeable. Clicking the **Check for MPAKs now** button causes an immediate check for received MPAKs. Also, on the actual network, the modem leaves its receiver on for a period of typically 10 seconds after any MPAKs are sent or received. This period is called transaction time, and allows the network to send additional MPAKs, or responses to previous MPAKs without introducing lengthy delays. Checking the file system frequently for a period of 10 seconds after sending or receiving an MPAK simulates transaction time. This state is indicated when Checking for 10 sec… appears in the **Active** field of the **Radio simulation control panel**. When multiple MPAKs are received, they are spaced about 1.5 seconds apart. On the actual network, received MPAKs can be closer together or further apart.
MPAK transmission	MPAK transmissions can be set to succeed or fail randomly (i.e. to always succeed, or to succeed only when approved by the user). To set the probability of successful transmission, adjust the slide bar located in the Radio simulation control panel dialog box. Successful MPAKs take about 1.5 seconds to complete, while failed MPAKs take about 3 seconds. Delays on actual networks can be much longer. When the success rate is manually set, the MPAK is pending in the modem until you click Yes or No in the dialog box asking whether or not the MPAK should succeed. This function is useful for simulating arbitrarily long delays, as the MPAK can be pending inside the radio for over 30 seconds under some network conditions.

Network feature	Description
Radio on / off	When the radio is turned off by the user, or `RadioStopReception` is called by an application, the simulator simulates the `Inactive` state on the network. In this state, all MPAKs sent to a modem are returned to the sender in a short period. Mailboxed MPAKs may, however, be placed in the mailbox. The recipient's radio simulator processes Inactive and Out of coverage MPAKs.
MPAK format checking	When sending MPAKs using the simulated Mobitex network, MPAK integrity is checked in much the same way as it is checked when sending MPAKs using an actual modem. The integrity of received MPAKs is also checked. This is to protect against incorrectly formatted MPAKs sent by custom host-side simulations.

Network features that are not simulated

The following network features, which are not typically used by applications in the Mobitex network, are not supported:

- Sendlist MPAK addressing
- Express mode support
- Grouplist addressing of broadcast MPAKs
- Personal subscriptions, login, logout and flexlist features
- SOS MPAKs

In addition, the following network behavior is not simulated.

Network feature	Description
MPAK received, but returned as undelivered	On the actual network with marginal wireless coverage, an MPAK could be returned as “Not sent” or “No transfer”, even though the receiver actually receives the MPAK. This behavior occurs when the MPAK is actually received, but the acknowledgement from the receiver does not reach the sender. As such, the sender concludes that transmission failed, while it was, in fact, successful. The simulator does not simulate this behavior.
Settings of SKIPNUM	On an actual modem, an application can set the value of SKIPNUM to change the amount of time before the radio modem communicates with the network from 10 seconds up to 160 seconds. In contrast, the simulator always checks for MPAKs at 10-second intervals, or when you click the **Check for MPAKs** button.
Invalid subscriptions	When operating a Mobitex modem with an invalid subscription, the network typically sends the modem a DIE MPAK, indicating that the modem is not allowed to send user data across the Mobitex network. In addition, invalid subscriptions, even before receiving a DIE MPAK, will not have their MPAKs forwarded. Instead, MPAKs are acknowledged and subsequently deleted by the network. Although invalid subscriptions should not happen, they have been known to occur on occasion. With the simulator, all subscriptions for which a simulator is running are assumed to be valid Mobitex subscriptions. Sending to simulators not running, however, is treated as though the MPAK was sent to an invalid subscription.

Writing a host-side simulation

When testing code running on the simulator against a server, you can write a host-side application that uses the file system to simulate Mobitex traffic.

The host-side simulation should be able to do the following:

- Format the raw MPAK.

  The MPAK must be formatted correctly. On the radio modem, the API formats the MPAKs header for you from the elements provided in the `DATA` structure. However, in this case, it is up to you to format the MPAK. See “Data packet structure” on page 51 for more general information on MPAKs.
- Create a temporary file containing the MPAK in the directory used by the simulator.
- Rename the temporary file to an available sequence number.

Sequence number checking can also be eliminated if your host is sending to that modem. In this case, it is safe to simply cycle through the sequence numbers 0 through 9.

In most cases, you do not need to set the **TIME** field, since the network overwrites it with its own time upon receipt. If you choose to fill in the **TIME** field, you must represent time as the number of minutes since the beginning of 1985. You can do this by using the `time` function and divide the returned value by 60 to get the number of minutes.

If the MPAK is incorrectly formatted, the receiving simulator displays a dialog box indicating what is wrong with the MPAK.

Checking for received MPAKs from a modem

To enable the simulator to send to your host, the host must indicate that it exists by creating the appropriate `.MAN` file. For example, if your host MAN is `12345`, it must create a file by the name of `00012345.MAN`. You can also manually create this file in the directory. In this case, if your host is not running, after the simulator sends 10 MPAKs they are returned due to congestion because no sequence numbers are available.

Your host-side application must check existing files for the MAN number periodically. You can use the Windows `_findfirst` mechanism to accomplish this. If a file addressed to your host implementation is found, you must read the file and subsequently delete it so that there is space for new MPAK files.

Example code for building a raw MPAK

```
#include <mobitex.h> // Contains the necessary constants.
static BOOL BuildMpak (MPAK_HEADER *header, BYTE *data,
  int data_length,BYTE *mpak, int *mpak_length)
{
  int data_offset;
  // Fill in fields for the application
  header->Sender = LocalMAN;

  data_offset = INDEPENDENT_HEADER_SIZE;
  data_offset += TIMESTAMP_LENGTH;
  if (header->MpakType == MPAK_HPDATA) {
    // HPDATA has one extra element in the header.
    ++data_offset;
  }
  // Place source MAN into MPAK. Stored as big endian
  mpak[0] = (char) ((header->Sender >> 16) & 0xff);
  mpak[1] = (char) ((header->Sender >> 8) & 0xff);
  mpak[2] = (char) (header->Sender & 0xff);
  // Place destination MAN into MPAK. Stored as big endian.
  mpak[3]=(char) ((header->Destination >>16)& 0xff);
  mpak[4]=(char) ((header->Destination >> 8)& 0xff);
  mpak[5]=(char) (header->Destination & 0xff);
  // Set packet header information
  mpak[PACKET_CLASS_OFFSET] |= PCLASS_PSUBCOM <<
    PACKET_CLASS_SHIFT;
  mpak[PACKET_TYPE_OFFSET] |= (char) (header->MpakType
    << PACKET_TYPE_SHIFT);
  if (header->Flags & FLAG_MAILBOX) {
    mpak[MAILBOX_FLAG_OFFSET] |= MAILBOX_FLAG_MASK;
  }
  if (header->Flags & FLAG_POSACK) {
    mpak[POSACK_FLAG_OFFSET] |= POSACK_FLAG_MASK;
  }
  if (header->MpakType == MPAK_HPDATA) {
    mpak[data_offset - 1] = (char) header->HPID;
  }
  // Put message into MPAK
  memcpy(mpak + data_offset, data, data_length);
  *mpak_length = data_length + data_offset;
}
```

5
Loading applications

This section provides information on how to load applications onto the radio modem.

Using the DLL utility

Before you load DLLs onto the radio modem, you can use the DLL utility to view statistics, such as the amount of flash memory and RAM that the DLL will require on the device.

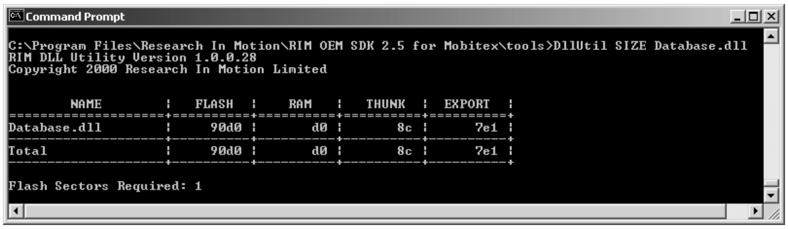
The DLL utility is `DllUtil.exe` in the SDK `tools` folder.

Use the following command to run the DLL utility:

```
DllUtil <SIZE [-R] | VERSION > <files>
```

`SIZE` displays on-device DLL sizes (this is the default command)
`-R` specifies that size calculations do not include relocation information
`VERSION` displays exported and imported APIs
`BATCH` allows you to specify a file that contains multiple files
`<files>` is one or more files (wildcards ***** and **?** are accepted)

The following illustration shows a sample DllUtil command:



```
C:\Program Files\Research In Motion\RIM OEM SDK 2.5 for Mobitex\tools>DllUtil SIZE Database.dll
RIM DLL Utility Version 1.0.0.28
Copyright 2000 Research In Motion Limited


        NAME        ¦   FLASH  ¦    RAM   ¦   THUNK  ¦  EXPORT  ¦
====================+==========+==========+==========+==========+
Database.dll        ¦      90d0 ¦        d0 ¦        8c ¦       7e1 ¦
--------------------+----------+----------+----------+----------+
Total               ¦      90d0 ¦        d0 ¦        8c ¦       7e1 ¦
--------------------+----------+----------+----------+----------+

Flash Sectors Required: 1
```

Using the program loader

The program loader utility (`Programmer.exe` in the SDK `tools` folder) loads compiled Windows .dll files onto the radio modem. You can also use the utility to manage applications that are already installed on the device.

Use the following command to run the program loader tool:

`PROGRAMMER [-P<port>] [-B<speed>] <command>`

`<port>` specifies the serial port number
`<speed>` specifies the serial port bit rate speed (such as 9600 or 115200).
`<command>` specifies the required action, one of: `HELP`, `BATCH`, `DIR`, `LOAD`, `ERASE`, `WIPE`, or `ALLOC`.

Note	This documentation provides information on the most commonly used commands. See the programmer.txt file, which is installed with the utility, for additional information.

Command line options

This section describes each command and available options.

LOAD

The `LOAD` command loads new applications or the application environment onto the device. Any old applications by the same name are erased. Applications should be grouped to conserve space on the device; however, if an application is being replaced separately from other applications on the device, it should be placed in its own group.

Note	When loading or replacing the application environment (`R902M.BIN`), it must be specified first on any `LOAD` command.

Usage **`LOAD [-S] [-G] <files or groups>`**

Options `-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 device.

`<files or groups>` are one or more files or groups of files to be loaded onto the device. Individual files are specified alone. Groups of files are enclosed in parentheses, brackets, or braces. Spaces must surround the brackets, as in the example below.

Examples

The following command loads the application environment and applications onto the device:

`PROGRAMMER LOAD R902M.BIN UI32.dll ( GPS.dll Interac.dll Transaction.dll Verify.dll )`

The following sample command loads a new credit card verification application, grouping it with the other applications:

`PROGRAMMER LOAD -G CreditCardVerify.dll`

Grouping programs

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

ERASE

The `ERASE` command removes applications currently loaded on a device. 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.

Usage

`ERASE –A or ERASE <application names>`

Options

`-A` specifies that all applications and the application environment should be deleted.

`<application names>` are the names of specific applications to be deleted.

Example The following command erases all applications on the device:

```
PROGRAMMER ERASE -A
```

The following command erases only the address book application:

```
PROGRAMMER ERASE ADDRESS.DLL
```

DIR

The `DIR` command lists the applications currently loaded on a device. 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 device.

Usage **`DIR [-S]`**

Options `-S` specifies a short listing of application names.

Example The following command lists the applications on a device:

```
PROGRAMMER DIR
```

BATCH

The `BATCH` command allows multiple commands to be placed in a file and executed with a single, short command. This command is useful when performing the same load process repeatedly.

The batch file might contain one or more of the `LOAD`, `ERASE`, `DIR`, or `BATCH` commands. The results are committed to the device only if all commands are completed successfully. Each line of a batch file can be at most 256 characters long, but single commands can be broken into multiple commands to accommodate long file names.

Usage **`BATCH <batchfile>`**

Options `<batchfile>` is the name of a file that contains commands.

WIPE

The `WIPE` command erases either the file system or the application region of flash memory. If no option is specified, both regions are erased. Otherwise, the specified region is erased.

You should only use this command when initially upgrading to the new application loader to ensure that old (unrecognized) versions of applications are destroyed.

Usage **`WIPE [-F | -A]`**

Options `-F` specifies that the file system should be erased.

`-A` specifies that the application area should be erased.

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

HELP

The `HELP` command invokes the built-in help system. With no options, a generic help message is produced. Help for a specific command can be obtained by specifying it as an option. Help for error messages can be obtained by specifying the `errors` option.

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.

Usage **`HELP [<command>]`**

Options `<command>` is the name of the command for which you want help.

Example For information on the `LOAD` command, type this command:

```
PROGRAMMER HELP LOAD
```

For information on errors, type this command:

```
PROGRAMMER HELP ERRORS
```

For a main help page, type this command:

```
PROGRAMMER HELP HELP
```

ALLOC

This command writes a new entry to the flash memory allocation log. The size of the File Area or the OS and App Area can be decreased only if all the sectors to be removed from these areas are completely blank. When the File Area size is decreased, at least one blank sector must remain within the area. You should back up your data before issuing the `ALLOC` command.

Usage **`ALLOC [-E] [-D <sectors>] [-A <sectors>]`**

Options `-E` specifies the File Allocation Sector to be erased before writing an entry.

`-D <sectors>` specifies the new size of the File Area (in flash memory sectors).

`-A <sectors>` specifies the new size of the OS and App Area (in flash memory sectors). If no option is specified, a report is given and no changes are made.

Example

```
PROGRAMMER ALLOC -E -D 16 -A 15
```

Troubleshooting

The following error messages require further explanation:

Error: Unable to connect to device

An error occurred trying to initiate communications with the device. Ensure that the device 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. Ensure that you have erased any old applications. If you have been erasing and loading applications often, you may have fragmented your flash memory space. In this case, try erasing all applications and starting again. Loading applications as part of the same group uses flash memory more efficiently.

Error: Relocation failed

It was not possible to relocate an application. You must specify the application environment (`R902M.BIN`) first in a `LOAD` command.

Error: Not all imports resolved

An application is requesting imports from another application that is not present. Ensure that you are loading all applications that provide the needed exports.

6 Radio communications

This chapter provides an overview of radio communications in the Mobitex network, including these topics:

- the structure of network layer data packets
- how to send and receive data packets

Note This chapter provides an introduction to radio communications on the Mobitex network. This information is intended as background information for developing radio modem applications. It is not intended to be comprehensive. Contact your network provider for complete documentation on network operation, including technical specifications on data packets.

Understanding the Mobitex network

Mobitex is a packet-switched, narrowband Personal Communications System (PCS) network designed for wide-area wireless data communications. Mobitex networks are operated by service providers such as Cingular Interactive in the United States, Rogers AT&T in Canada, and other companies in Asia, Australia, Europe, and South America.

Wireless applications typically send short amounts of data in bursts, with fairly long delays between each transmission. Packet switching uses limited radio frequency resources efficiently by enabling multiple users to share channels.

Mobitex provides highly reliable, 2-way digital data transmission. The network provides error detection and correction, including transmission acknowledgement, to ensure the integrity of the data being sent. The network's packet-switching technology provides great flexibility and efficiency for wireless data transmission, especially when the application involves messaging, dispatching, remote queries, or other situations in which only small amounts of data are transferred.

Routing

All data exchanged in a Mobitex network is contained in MPAK packets.

Each device on the Mobitex network is assigned a unique, 24-bit Mobitex access number (MAN) to identify the device. Messages are routed through the network from sender to receiver in the form of packets.

Before sending MPAKs, an application must register for radio events by calling `RadioRegister`.

The application itself can assemble and format MPAKs, or it can fill in elements on the MPAK header structure and send it, along with the data, to the radio modem application server for assembly into an MPAK.

Sending packets

To send a packet, an application calls `RadioSendMpak` and includes the information and length of the data to send. A tag number is returned for each packet submitted to the radio. After the packet has been transmitted and acknowledged by the Mobitex network, the RIM 902M Radio Modem application server returns a `MESSAGE_SENT` event and tag number to the application that sent the packet.

If the packet could not be delivered to the Mobitex network (for example, the radio was out of wireless network coverage), the application receives the notification message `MESSAGE_NOT_SENT` along with an error code and tag value number. Refer to “Radio API error codes” on page 129 for a complete list of errors.

If the packet reaches the Mobitex network but the destination is unreachable, the packet is returned to the radio modem and is handled like a newly received packet. These packets have the `TRAFFIC_STATE` element of the `MPAK_HEADER` structure set, which indicates the error code.

You can stop MPAK transmission using `RadioCancelSendMpak`.

Note Cancelling an MPAK does not guarantee that it was not received before it was cancelled.

The radio modem can store up to 4 MPAKs internally pending for transmission. If an application wants to send a small number of MPAKs, it can call `RadioSendMpak` repeatedly without waiting for previous MPAKs to be sent. If an application wants to send many MPAKs, it must send the MPAKs one at a time.

Receiving packets

To receive packets from the network, an application must be registered to receive `RADIO` events (using `RadioRegister`). When packets are received from the Mobitex network, all registered applications are notified through the `RADIO` device, `MESSAGE_RECEIVED` event. Each notified application must retrieve the packet by calling `RadioGetMpak` before yielding control to the system. After all registered applications have received the message, the MPAK is released by the system.

If an application cannot store an MPAK that it has received, the application might return the MPAK to the system using `RadioStopReception`. `RadioStopReception` allows applications to cancel reception of MPAKs; however, reception is blocked for all applications, and the modem informs the network that it is no longer ready to receive MPAKs. The effect of `RadioStopReception` is reversed by calling `RadioResumeReception`. At this time, the stored MPAKs are received again, and the modem informs the network that it is ready to resume receiving packets.

Data packet structure

MPAKs are network-layer data packets. An MPAK structure contains the following information:

- 24-bit sender MAN
- 24-bit addressee MAN
- 24-bit time stamp of 1-minute intervals
- 2 bytes of type information
- up to 512 bytes of payload data

If it is necessary to send more than 512 bytes of data, the application must divide the data into two or more MPAKs.

There are many types of MPAKs defined in the Mobitex interface specification (MIS), but many are not supported or not widely used. Contact your network provider for complete details on MPAK types defined for the Mobitex network.

MPAK header format

This section summarizes the format of MPAK headers, as they are packaged inside an MPAK.

The RIM 902M Radio Modem API normally assembles and disassembles the header, unless formatting is bypassed by sending and receiving raw MPAKs. A raw MPAK has a fixed-format 8-byte header containing the source, destination, type, and either the flags or the traffic state. On most MPAKs, this header is followed by a 24-bit time stamp, and up to 512 bytes of data.

All MPAKs start with a header. The following table describes the header format.

Bytes	Description
3	Source MAN in most significant bit (MSB) to least significant bit (LSB) format This field tells the network who sent the MPAK, and where to send a failure status or positive acknowledgment (if any).
3	Destination MAN (in MSB to LSB format) This field tells the network where to send the MPAK.
1	Bitfield indicating status for incoming and procedure for outgoing MPAKs The following bits tell the network how to handle a sent MPAK (1 = True and 0 = False): • Bit 0 = Use mailbox • Bit 1 = Return positive acknowledgement • Bit 2 = Use address list • Bits 3 to 7 should be set to 0 The following bits indicate the state of a received MPAK:
1	MPAK type This field determines which kind of MPAK this is.

22	Address list This field is optional, and used only if the Use address list bit is set.
3	Timestamp Use this field with `TEXT`, `DATA`, `STATUS`, and `HPDATA` MPAK types.
0 to 512	Data The format of this field depends on the MPAK type.

Source MAN

The source MAN identifies the device that originated the MPAK. The RIM 902M Radio Modem API will automatically set this MAN to its own MAN when sending an MPAK.

Destination MAN

The destination MAN identifies the device that is the intended recipient of the MPAK. To send an MPAK, your applications must know the MAN of the destination device. When replying to a message, applications can read the source MAN from the header of the received MPAK and use that to send a reply.

The `RadioSendMPAK` function can contain a pointer to an `MPAK_HEADER` structure, in which `header.Destination` contains a `DWORD` indicating the destination MAN.

If, for the purpose of reporting failure status or delivery confirmation, the MPAK is returned to the sender, the source and destination MANs do not change. The returned MPAK is an exact duplicate of the original MPAK that was sent, with the exception of the traffic state bits indicating its failed status.

MPAK flags

MPAK flags can be set when sending an MPAK to indicate to the network that special handling is required. If no special handling is required, then the flags must all be set to zero.

The `header.Flags` component of the `MPAK_HEADER` might consist of the various MPAK flags OR'ed together. Valid flags that can be used in `RadioSendMpak` are `FLAG_MAILBOX` and `FLAG_POSACK`. The address list flag is not supported, so the application must build its own MPAK header if an address list is to be used.

The header.TrafficState component must always be set to TS_MESSAGE_OK when sending an MPAK.

MPAK traffic state

When receiving an MPAK, the MPAK status indicates what happened to the MPAK as it was transmitted through the network. The status also indicates whether the MPAK is being returned to the sender.

This value is contained in the header.TrafficState component when the RadioGetMpak function is called. It must always be set to TS_MESSAGE_OK when sending an MPAK.

MPAK type

The MPAK type byte determines which kind of MPAK is being sent or received. The structure and meaning of the rest of the MPAK depends on its type. The MIS defines many different MPAK types; however, some of these are for private use between the modem and the network, and some are not supported. This document only deals with MPAKs that are required for applications using the Mobitex network.

This value is contained in the header.MpakType component of the MPAK_HEADER structure, for both sent and received MPAKs.

MPAK timestamp

Each MPAK is given a time stamp when it is received on the Mobitex network. This applies to the TEXT, DATA, STATUS, and HPDATA MPAK types.

You do not need to set the TIME field when sending an MPAK, because the network automatically sets the Time field when it receives it. You can fill these three bytes with 0.

The TIME value stored in the MPAK is the number of minutes since 12:00 A.M. January 1, 1985, Mobitex Local Time. The RIM 902M Radio Modem API automatically decodes time stamps in received MPAKs, and places the information in the MPAK_HEADER structure.

When an MPAK is received, the header.lTime element contains a 24-bit value that represents the raw 24-bit Mobitex time. The header.TimeStamp element contains this same time, converted to a TIME structure.

Sending an MPAK to multiple destinations

Mobitex allows a mobile device to send data to more than one destination with a single MPAK transmission. This can be accomplished in three ways:

- address list
- group MAN
- personal subscription MAN

Address list

The `Address list` bit allows a radio modem to specify up to seven distinct destinations with a single transmission. None of the destinations receive the actual address list or any indication that the message was sent to an address list.

To use an address list, set the `USE ADDRESS LIST` MPAK flag to 1. This feature can be used only when sending an MPAK of type `TEXT`, `DATA`, `STATUS`, or `HPDATA`. The MPAK destination MAN must be set to the value 0x000001, which is the MAN of the Mobitex network.

An address list immediately follows the MPAK type byte. The address list is always 22-bytes long, regardless of the number of destinations. The first byte, `Count`, must be a value from 1 to 7. MANs are stored in order of most significant byte (MSB) to least significant byte (LSB). The unused MANs must be set to 0. The list has the following format:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
Count	MAN #1			MAN #2			MAN #3			MAN #4			MAN #5			MAN #6			MAN #7		

Since address lists are used very rarely, the RIM 902M Radio Modem API does not support the building of these types of MPAKs. Instead, the application must format the MPAK and send it as a raw MPAK.

Group MANs

To use group broadcasts, special arrangements must be made with the network operator. The network operator assigns a list of MANs to a special group MAN. Sending an MPAK to the group MAN causes a copy of the MPAK to be sent by the network to each MAN in the group's list. Each radio modem can belong to a maximum of 15 groups.

Personal subscription MANs

A personal subscription MAN (PMAN) can be moved by the network from one mobile device to another, which enables users to choose from a pool of mobile devices. To use personal subscription MANs, special arrangements must be made with the network operator. The RIM 902M Radio Modem does not support personal subscriptions.

Traffic state bits

The traffic state bits describe what happens to a received MPAK as it is passed through the network. These bits are meaningful only when receiving an MPAK, not when sending one.

Note For traffic states `OK` and `From_mail`, the destination MAN listed in the MPAK header is your device's MAN and the source MAN is the sender's MAN. For all other traffic states, the MPAK is returned to your device with the source and destination MANs unchanged. The source MAN is yours and the destination MAN is that of your original, intended recipient.

The following traffic states apply to received MPAKs:

Binary value	Meaning	Description
000xxxxx	`OK`	This packet arrived from the source without encountering any problems.
001xxxxx	`From_mail`	This packet was stored in your Mobitex mailbox until you were able to receive it.

The following traffic states apply to returned (undelivered) MPAKs:

Binary value	Meaning	Description
010xxxxx	`In_mail`	This packet, in which the Use_Mailbox flag was turned on, is in the destination's mailbox.

011xxxxx	`No_transfer`	This packet could not be delivered to the destination.
100xxxxx	`Illegal`	This packet violated a Mobitex network requirement. You may have sent a packet to a group MAN or a fixed terminal with the `Return_Positive_Ack` flag set.
101xxxxx	`Congest`	This packet could not be sent through the network because the network channels are congested.
110xxxxx	`Error`	Your transmitted packet could not be sent through the network due to a technical error.
111xxxxx	`Busy`	Your packet could not be delivered because the recipient is busy with a real-time connection (not normally used).

Delivery confirmation

When the `Return positive acknowledgement` bit is set, the Mobitex network returns a complete copy of the MPAK to the source radio modem when the destination receives the MPAK. This bit is set only when sending a TEXT, DATA, STATUS or HPDATA MPAK type.

This flag cannot be used when sending to a group MAN, to the network (MAN 0x000001), or to a fixed FST terminal, such as a server connected to the Mobitex network through an X.25 connection. The network returns the packet with a traffic state of `Illegal`.

Copies of MPAKs returned to the sender after successful deliveries are identical to the actual MPAK that was sent, including the traffic state bits.

Mailbox

If your application sends an MPAK with the `Use mailbox` flag set and the destination device is not in wireless network coverage, the network stores the MPAK in a mailbox. Your application receives a copy of the MPAK with a traffic state of `In_mail`. After the destination re-establishes network contact, the contents of the mailbox are sent to the modem, with a traffic state of `From_mail`.

If both the `Return positive acknowledgement` and the `Use mailbox` flags are set, the network sends a copy of the original MPAK to the sender after the MPAK is delivered to the device. (This is in addition to the MPAK originally returned to the sender with a traffic state of `In_mail`.) The returned MPAK indicates to the sending device that the destination device has returned to coverage and has received the MPAK.

The network can store a small number of MPAKs for each modem. If the network operator has not configured the destination's subscription to include the mailbox option, the network will ignore the mailbox flag.

User-data MPAK types

There are many different MPAK types that are useful on the Mobitex network. The most useful MPAK types are the user-data types: TEXT, DATA, STATUS, and HPDATA. These are the only MPAKs that are used for transferring data on the Mobitex network, and are likely the only MPAKs that your application uses. In fact, most applications typically use just one MPAK type and ignore all others.

All application-level communications take place with user data MPAKs. When sending MPAKs, you can have the RIM 902M Radio Modem API automatically format the MPAK and send it. Simply fill in the appropriate elements in the `HEADER` structure and call the `RadioSendMpak` function.

These MPAKs require the 3-byte `Timestamp` field. When sending an MPAK, this field is set to 0x000000. The radio modem's API sets this field to 0 automatically when it sends an MPAK.

These MPAKs can use the optional `Address list` field to send data to multiple destinations.

If multiple applications are running on the RIM 902M Radio Modem, every application receives notification of a received MPAK. This means that one application can receive an MPAK that is intended for another application. Therefore, you should structure your MPAK data field so that it is uniquely identified as belonging to your application. For example, you can use a unique HPID (Higher Protocol Identifier) or you can include a unique sequence of bytes in the start of your data field, and reject MPAKs that are not structured in this way.

TEXT

This MPAK type contains 1 to 512 bytes of user data in text form. Each network operator defines a set of accepted characters. Sending a TEXT MPAK with bytes that are not in the accepted character set causes the entire MPAK to be returned with a Traffic state of `illegal`. For this reason, most applications do not use this MPAK type.

MPAK Type	Value	Send or Receive	Format of data area
TEXT	0x01	Send / Receive	User data (1 to 512)

DATA

This MPAK type contains 1 to 512 bytes of user data. All byte values are accepted. Since there is no defined standard for the format of a DATA MPAK, the DATA MPAK is best used in applications where the destination and sources of DATA MPAKs are well known and controllable.

MPAK Type	Value	Send or Receive	Format of data area
DATA	0x02	Send / Receive	User data (1 to 512)

STATUS

This MPAK type contains one byte of user data. There is no defined standard for its meaning.

MPAK Type	Value	Send or Receive	Format of data area
STATUS	0x03	Send / Receive	User data (1)

HPDATA

This MPAK type contains 1 to 512 bytes of user data. All byte values are accepted. The HPID byte specifies the format of the user data, including transport protocol, compression algorithm, and encryption scheme. HPIDs 128 to 199 and 221 to 255 are free for use. All other values are reserved for existing and future protocol standards.

MPAK Type	Value	Send or Receive	Format of data area
`HPDATA`	0x04	Send/Receive	HPID number (1) User data (1 to 512)

Other MPAK types

This section describes MPAK types that your application is not likely to require, but might receive during the course of normal operation. All MPAKs described in this section are used by Mobitex modems and networks to coordinate their interaction.

All specific handling that might be required for other miscellaneous, non-user data MPAK types is already handled by the radio modem, so you do not have to process or format those types. Copies of these received MPAKs are passed on to all applications. Since you are not likely to use them, you can design your application to ignore them.

Note The RIM 902M Radio Modem API does not support assembling and disassembling of these MPAK types. Instead, applications that want to process these MPAKs must process them as raw MPAKs.

ACTIVE

This MPAK is used to notify the network of the modem's presence, without actually sending any user data to the network. If the modem has data to send to the network, the network automatically detects that the modem is within range of the network.

If no MPAKs have been sent by any applications in the interval, the RIM 902M Radio Modem automatically sends an `ACTIVE` MPAK five seconds after the radio is turned on and in wireless network coverage. The network then releases any MPAKs in the mailbox, and subsequently delivers messages intended for the device.

MPAK Type	Value	Send or Receive	Format of data area
`ACTIVE`	0xC7	Send to MAN 0x000001	0x00000000 (4 zero bytes)

INACTIVE

When the network receives an `INACTIVE` MPAK, it marks your device as inactive, indicating that it is turned off or out of wireless network coverage. It is not able to receive any MPAKs until the device is turned on or back in coverage. MPAKs intended for your device are returned to the sender as undeliverable.

The mobile device can also send an `INACTIVE` MPAK to the network as a means of flow control. Applications should not directly send `INACTIVE` MPAKs to the network when they can no longer store messages. Instead, the high-level function `RadioStopReception` should be called, which results in an `INACTIVE` MPAK being sent.

MPAK Type	Value	Send or Receive	Format of data area
`INACTIVE`	0xC8	Send to MAN 0x000001	Empty

GROUPLIST

This MPAK lists the group MANs to which the radio modem belongs. The first byte in the data area is the actual number (N) of MANs in the list. This is followed by exactly 15 MANs. Only the first N of these contain actual MANs; the rest are filled with zeros. Your application should ignore this MPAK unless you specifically use group MANs.

MPAK Type	Value	Send or Receive	Format of data area
`GROUPLIST`	0xCF	Receive	Number of MANs in list (1) 15 personal subscription MANs PMANs (3 each, total 45)

TIME

The `TIME` MPAK contains the current Mobitex Local Time, at the time when the packet was issued by the network. A `TIME` MPAK is broadcast by the network every ten minutes, but most devices in low-power mode (the default state) do not have their receivers turned on and do not receive the `TIME` MPAK.

MPAK Type	Value	Send or Receive	Format of data area
`TIME`	0xD4	Receive	Time (3)

DIE

The `DIE` MPAK is sent by the network to unregistered or illegal devices. After the device has received a `DIE` frame, it can no longer transmit any user data MPAKs.

MPAK Type	Value	Send or Receive	Format of data area
`DIE`	0x09	Receive	Empty

LIVE

This MPAK, the opposite of `DIE`, returns the device to normal operation.

MPAK Type	Value	Send or Receive	Format of data area
`LIVE`	0x10	Receive	Empty

ROSI blocks

Due to the fluctuating signal levels common in wireless data networks, a high bit error rate means that an MPAK cannot be sent reliably over the air in one burst.

As a result, each MPAK is further divided into smaller units of data, called radio-oriented synchronous information (ROSI) blocks, for transmission between the radio modem and the network base station at the data link layer.

Each ROSI block carries 18 bytes of actual payload data, although the first ROSI block has 6 bytes of overhead. As such, a raw MPAK that fits into 12 bytes can be sent in one ROSI block.

An MPAK can be split into a maximum of 31 ROSI blocks, which are transmitted together over the air as one ROSI frame. If the base station is unable to decode some of the blocks, it requests retransmission of specific individual ROSI blocks. This is more efficient than resending an entire MPAK.

When transmitted across the air with error correcting code, each ROSI block is 240 bits in length, with 144 bits of payload data. ROSI blocks are transmitted at 8000 bps. As such, transmitting a maximum length MPAK takes nearly one second. However, given the access protocol, sending or receiving a maximum length MPAK typically takes 1.5 to 2 seconds in an area of good wireless network coverage. Power consumption while using the radio is substantial, and should always be taken into consideration. For example, the RIM 902M Radio Modem consumes three orders of magnitude more power while sending an MPAK than it does under normal operation.

7
Understanding the RIM 902M Radio Modem

This chapter explains key aspects of the radio modem operating system:

- operating system messaging
- file system
- serial communications

Operating system messaging

The RIM 902M Radio Modem API supports multiple applications running on the device simultaneously, as well as multi-threaded applications. Each application on the modem receives an execution thread at startup. Applications can create and destroy additional threads dynamically.

The operating system employs co-operative multitasking, so that no application can preempt another application in midstream unless the first application explicitly yields control.

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 radio modem does not appear to lose functionality.

A messaging system facilitates communication between the operating system and radio modem threads. Applications receive messages that describe events and their associated parameters, and can also send messages for other threads to process.

Messaging

Applications written for the RIM 902M Radio Modem receive all external notification through events sent to the applications. In addition to receiving messages from the system, applications with multiple tasks can also exchange messages between threads.

After processing an event, applications call the `RimGetMessage` function to receive the next event. If no event is available, the application yields to allows other applications to run. If no other applications have events to process, the CPU is put in a standby state until the next event, such as the expiration of a timer, initiates the application again.

`RimGetMessage` can be called from anywhere in an application. An application can retrieve and process messages differently depending on the state of its execution.

In some cases, the application might want to process certain events in the same way regardless of the application's state. The application can call the function `RimRegisterMessageCallback` to register callbacks for message processing. `RimRegisterMessageCallback` registers a function to be called on specific types of messages. These functions are called when the main part of the application is blocked on `RimGetMessage` to avoid potential concurrency problems.

In addition to receiving messages from the system, applications can also exchange messages with each other.

Note When one application calls another application, the task ID is still that of the calling application. If anything causes a message to be sent subsequently, the message is sent to the calling application, rather than the called applications.

When sending messages between applications, the `Device` field may or may not be set to one of the system devices. If one of the system devices is specified, the protocol defined there should be maintained to avoid causing problems for other applications.

There are two ways to send messages from one task to another: asynchronous and synchronous.

Asynchronous send

For asynchronous communications (non-blocking send), applications post messages to another application's message queue by calling the API function `RimPostMessage`. The destination application receives the message after previous events in its message queue are processed. The sending process continues execution immediately after the call to `RimPostMessage`.

Synchronous send

For synchronous communications (blocking send), applications send the message to another application's message queue by calling the API functions `RimSendMessage` or `RimSendSyncMessage`. `RimSendMessage` is provided for compatibility reasons and can be removed at a later time. The sending application calls `RimSendSyncMessage` and waits until the destination process replies using `RimReplyMessage`. The destination application processes all events in the order in which they are received unless `RimWaitForSpecificMessage` is used. The destination application then processes the message, which might involve calling other functions (including `RimTaskYield`). Although the operating system might be able to detect deadlock, take precautions when using synchronous send because tasks are blocked until the receiving application replies to the message. It is possible that the sending application's message queue could become full if the message is not replied to quickly. To prevent this, the sending process should use `RimSetReceiveFromDevice` or `RimToggleMessageReceiving`.

Synchronous messages are not sent if the task ID is invalid or if the destination task is waiting for another task to receive a message. In either case, the `RimSendSyncMessage` function returns an error immediately.

Understanding the file system

The radio modem stores permanent data in flash memory.

The file system provides an abstract model of a database, with a sequence of variable-sized records.

Your application design should take into consideration the unique characteristics of the radio modem file system. In particular, the file system is best suited for applications that read data frequently but change data less often and use many smaller data items rather than large data items.

Database and streamed file models

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

You can create orphan records. Orphan records do not belong to any database, but can be made a part of 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.

All file system function names begin with `Db`, and those pertaining to streamed files begin with `DbFile`.

Reading data

Reading from flash memory occurs at a speed comparable to Dynamic RAM (DRAM), which is limited only by the CPU clock rate in the radio modem.

Using pointers to retrieve data

The file system uses handles to uniquely identify every database and record object. These handles are temporary; they persist for the lifetime of the item only on a particular system.

Note You should not store handles (or pointers) in permanent data, because this data is not portable to other systems.

Handles are represented as 16-bit unsigned integers.

The file system maintains a system-wide handle lookup table for all records. Applications can read directly out of the flash memory that is used to store the files.

The `DbPointTable` function returns the address of the handle table. Indexing the table with a handle yields a pointer to data. For a database handle, the data represents the database directory entry. For a record handle, the data is the actual record data. Currently, the table has 250 pointers, which limits the number of records to less than 250.

Applications can cache pointers, but they must re-read pointer values after the application writes to the file system or yields control to another process. Pointers are not guaranteed to persist across any operation that might change the file system’s permanent data or its organization.

You can achieve 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.

Debugging pointer errors

Applications must re-read pointers after an application writes data to the file system or yields execution to another application.

To help trap this error quickly, the simulator periodically moves the entire file system, marks 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 may subsequently point at regions of memory that are now invalid.

The affected code causes a page fault to occur, so you can use Microsoft Visual Studio to find these errors quickly.

You can disable this behavior using the `/T-WY` command line option.

Writing data

Writing data to flash memory involves saving the contents of the entire 64-KB flash memory sector, erasing the flash memory sector, and rewriting the contents of the entire flash memory sector with the changes. As a result, the file system generally does not modify flash contents in place. Instead, it employs a log file system with the following features:

- Write operations are performed sequentially at the end of a continuously growing log.
- When an existing data item is to be modified, a new copy is always written at the end of the log. The old copy is marked as changed and left in the log.
- Periodically, when the file system runs out of space, some flash memory sectors are cleaned up. The valid data is copied elsewhere and sectors are erased.

Write verification

All changes to the file system's permanent data or its organization are verified by reading the data after it has been written completely to flash memory. If any mismatch is detected, the entire 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 does not verify that the remaining bits are unchanged.

Atomic changes

Almost all operations that change the permanent data, such as creating or modifying records, are atomic: a record is either written completely to the database, or the database remains unchanged. No intermediate state exists. This preserves the consistency of the data if the system crashes for any reason during the operation.

Note The `DbAndRec` function and streamed file access functions (which start with `DbFile`), are not atomic. Refer to the individual function descriptions for specific details.

Serial communications

The serial communications API enables you to configure serial port parameters and to use data signal ready (DSR) / data terminal ready (DTR) signalling. The RIM 902M Radio Modem also has the capability to control hardware flow with OS support.

The *RIM* 902M *Integrator Guide* provides pinouts for the serial ports.

Bidirectional I/O Lines

The RIM 902M Radio Modem has four bidirectional input / output (I / O) lines for developers to use. These I / O lines are available at pins 1 through 4, and are controlled using the standard C++ functions `_inp()` and `_outp()` in the `conio.h` header file.

The controlling port addresses are as follows:

- `0xF861` - Pin Direction Register
- `0xF862` - Port Data Latch Register
- `0xF863` - Pin State Register

Only the lowest four bits of data sent to, or read from, each of these ports is significant. The most significant bit corresponds to pin 1.

The Pin State Register reads the logic state of the port pins.

When the Pin Direction Register bit is set to 0, the corresponding I / O pin is driven strongly to 0 or 1 depending on the value in the Port Data Latch Register. If the Pin Direction Register bit is set to 1, the pin can be used as a high-impedance input or an open-drain output.

Set a Port Data Latch Register bit to 1 to use the pin as an input or to float the open drain output. Set the bit to 0 to drive the open drain output to 0.

Pin Direction Register	Port Data Register	Result
0	0	Force pin to GND
0	1	Force pin to 3V
1	0	Force pin to GND
1	1	Use as input or float pin

I^{2}C Support

The RIM 902M Radio Modem includes an I^{2}C (Inter-IC bus) module on the main board, providing an interface between the radio modem and peripheral circuits. The I^{2}C module also adds several other features, including analog-to-digital and digital-to-analog converters.

Refer to “Sample I^{2}C driver” on page 169 for an example of the code.

8
System API

This chapter provides information on System API common structures and functions.

The System API provides a wide range of functionality to radio modem applications, including thread management, message handling, and task switching.

System API structures

The System API uses the following structures.

MESSAGE structure

Any system or application message is in the form of a `MESSAGE` structure (defined in the header file `Rim.h`):

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

Field	Description
Device	This mandatory field describes the sender of the message. If the sender is a radio modem device, the device field of the message is set to one of: • DEVICE_SYSTEM • DEVICE_RTC • DEVICE_TIMER • DEVICE_SERIAL • DEVICE_RADIO Refer to “Device events” on page 159 for a description of each device and its events.
Event	This mandatory field describes the event that caused this message to be sent.
SubMsg	This optional field contains extra data pertaining to the event.
Data	This optional array contains extra data pertaining to the event.
DataPtr	This optional field provides the ability to pass large amounts of data through a reference external to the message itself.
Length	This optional field specifies the record size referenced by `DataPtr`, if used. This field can be used to pass extra data if necessary.

It is recommended that the `Device` and `Events` fields be set; however, the RIM 902M Radio Modem API does not examine or validate messages sent between applications.

TIME structure

The `TIME` structure (defined in the `Rim.h` header file) represents an instance of time for applications to retrieve or set the date and time.

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

Field	Description
`second`	Seconds (0 to 59)
`minute`	Minutes (0 to 59)
`hour`	Hour (0 to 23)
`date`	Day (0 to 31).
`month`	Month (1 to 12)
`day`	Day of week (0 to 6; Sunday = 0, Saturday = 6)
`year`	Year (1998-2090)
`TimeZone`	Time zone. GMT is 0; all other zones are counted in half-hours from that point. For example, EST is GMT –5 hours which would be stored as –10 half-hours.

System API functions

RimCatastrophicFailure

Handles unrecoverable application errors

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

Parameters: `FailureMessage`

A pointer to a string to be output to the debug stream describing the failure

Return Value: No return value

Remarks: This function halts the radio modem and is used to handle unrecoverable application errors that can be cleared only by resetting the system.

Example: Refer to `GPS.C`.

RimCreateThread

Dynamically creates a new thread, with its own stack, in the application server

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

Parameters: `Entry`

A pointer to the entry function

`Stacksize`

This parameter is the size, in bytes, of the local stack for the newly created thread. This value must be large enough to hold the stack for the thread, as well as the stack space required by the API when called. Setting this value too small creates unpredictable results. It is recommended that this value never be set less than 2,000.

Return Value: This function returns the task handle of the newly created thread. Zero is returned if the thread could not be created.

Remarks: This function enables an application to create a new thread dynamically, with its own stack, in the application server. The new thread shares the same data space as the parent process. If there are no more task handles available or not enough memory, the function fails.

Created threads cannot be placed in the foreground unless the thread enables this attribute with the `RimSetPID` function.

Threads do not receive any radio events unless they register for them using the `RadioRegister` function.

Example: Refer to `GPS.C`.

RimDebugPrintf

Prints formatted text to the debug stream

```
void RimDebugPrintf( const char *String, [arg] ...)
```

Parameters: `String`

A format control string, as used in the standard library function `printf`

Return Value: No return value

Remarks: When called in the Windows environment, the `RimDebugPrintf` function formats and prints a series of characters and values to the debug output stream, which can be displayed in the Output window of Developer Studio. If arguments follow the format string, the format string must contain the appropriate output specifiers.

When called on the radio modem, `RimDebugPrintf` has no effect.

Example: Refer to `GPS.C`.

RimFindTask

Searches for a task based on its version string

```
TASK RimFindTask( char *pPrefix )
```

Parameters: `pPrefix`

The prefix of the version string of the task for which to search

Return Value: This function has a return value of the handle of the task that has a version string that starts with `pPrefix`. If no such task is found, `RimFindTask` returns `TASK_NOT_FOUND`.

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

RimFree

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

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

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

Return Value: No return value

Remarks: This function de-allocates memory from the far heap and returns the block to the free list.

Note Attempting to free a block not allocated by `RimMalloc` can result in subsequent failure when allocating memory.

RimGetAlarm

Finds the earliest enabled alarm set by any application

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

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

Return Value: This function returns the handle of the task that has set the earliest future alarm date and time using `RimSetAlarmClock`. If no alarms are enabled, `RimGetAlarm` returns 0.

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

RimGetBatteryStatus

Reports the status of the power supply voltage.

```
DWORD   __cdecl RimGetBatteryStatus(void);
```

Return Value: One of the following flags indicating the current battery status:

`BSTAT_TOO_HOT`	`0x20000000`	Device is too hot
`BSTAT_LOW`	`0x10000000`	Supply voltage is too low

Remarks: When either of the flags above is set, the `BSTAT_NO_RADIO` flag is also set to indicate that the radio is unable to transmit.

RimGetCurrentTaskID

Gets the current task ID

```
TASK RimGetCurrentTaskID( void )
```

Return Value: The task handle of the currently running task

Remarks: This function returns the handle of the currently running task. This task handle is the same as that received by a task on the `INITIALIZE` and `POWER_UP` events from the application server.

RimGetDateTime

Gets the date and time from the real-time clock

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

Parameters: `Time`

A pointer to a `TIME` structure.

Return Value: No return value

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

The earliest time that can be returned by `RimGetDateTime` is `01/01/1998 00:00:00`. The latest time that can be returned by `RimGetDateTime` in the embedded system is `12/31/2090 23:59:59`.

The Windows simulator, however, uses `mktime` which can only handle a date from January 1, 1970 to midnight, February 5, 2036. Consequently, `RimGetDateTime` in the Windows simulator cannot return any date later than February 5, 2036.

RimGetMessage

Obtains the next message from the application message queue

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

Parameters: `Msg`

A pointer to a `MESSAGE` structure that receives the message

Return Value: The task handle of the task that sent or posted the received message

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

It is important that a task calls `RimGetMessage` to obtain events, as this enables the application server to determine which tasks are currently not using the CPU. The processor can then be put into low power mode. If `RimGetMessage` is not called, all other applications remain blocked and the radio modem becomes unresponsive.

Example: Refer to `SERIALDEMO.C`.

RimGetMessageRaw

Obtains the next message from the application message queue without triggering callbacks

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

Parameters: `Msg`

A pointer to a `MESSAGE` structure that receives the message

Return Value: The task handle of the task that sent or posted the received message received

Remarks: The functional semantics of `RimGetMessageRaw` are identical to that of `RimGetMessage`. However, this function enables the calling application to obtain the next message from its message queue without triggering any registered callbacks (refer to `RimRegisterMessageCallback`). `RimGetMessageRaw` can be called from within a registered callback function.

RimGetOSversion

Gets the operating system version

```
DWORD RimGetOSversion( void )
```

Return Value: This function returns the operating system version. Version information is packed into the `DWORD` return value as follows:

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

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

RimGetPID

Retrieves attributes of other processes

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

Parameters: `HTask`

This parameter is the task number for which information is desired. Valid values are 0 (the system task) through –1 (`MAX_APPLICATIONS`).

`Pid`

A pointer to the `PID` structure to be filled in with information about the specified task

`Subtitle`
The address of the pointer to be set to point to the specified task's subtitle string

Return Value: `TRUE` if the task exists or `FALSE` if the task number is invalid

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

RimGetTicks

Retrieves the amount of time passed since the radio modem was turned on

```
long RimGetTicks( void )
```

Return Value: This function returns the number of time ticks, in 10-millisecond increments, since the radio modem was turned on. A value of 5000, for example, indicates 50 seconds.

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

Example: Refer to `SERIALDEMO.C.`

RimInitiateReset

Causes the radio modem to reset

```
void RimInitiateReset( void )
```

Return Value: No return value

Remarks: This function causes the radio modem to reset.

RimKillTimer

Cancels a timer that was previously set by the application

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

Parameters: `TimerID`

The identifier of the timer to be cancelled

Return Value: No return value

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

RimMalloc

Allocates memory

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

Parameters: `Size`

The size of the block of memory to be allocated

Return Value: The `RimMalloc` function returns a void pointer to a block of allocated memory. It returns `NULL` if sufficient memory is not available.

Remarks: The memory block allocated might be larger than the size requested due to alignment and system overhead. A requested size of 0 returns a valid pointer to a zero-length block. Applications should always check the return value of this function.

RimMemoryRemaining

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

```
DWORD RimMemoryRemaining( void )
```

Return Value: The number of free bytes in the heap

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

RimPeekMessage

Determines whether there are application messages to be processed

```
BOOL RimPeekMessage( void )
```

Return Value: `TRUE` if there is a message waiting on the event queue for the calling task, otherwise, `FALSE`

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

RimPostMessage

Posts a message to another application

```
void RimPostMessage( TASK HTask, MESSAGE *Msg )
```

Parameters: `HTask`

A handle to the application to which the message is being sent

`Msg`

A pointer to a `MESSAGE` structure

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

RimRealloc

Reallocates memory

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

Parameters: `Ptr`

A pointer to the memory block to grow or shrink

`Size`

The desired new size of the block

Return Value: The `RimRealloc` function returns a void pointer to a block of allocated memory. It returns `NULL` if a block of the requested size could not be created.

Remarks: The block may be larger than the size requested due to alignment and system overhead. A requested size of 0 returns a valid pointer to a zero-length block. Applications should always check the return value of this function.

The system may move the contents of the old block of memory to find the requested space, if necessary. A return value of `NULL` means that a block of the requested size could not be created; the existing block of memory (as referenced by `block`) is untouched in this case. If `block` is `NULL` when this function is called, the function acts as `RimMalloc`.

RimRegisterMessageCallback

Registers a callback function

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

Parameters: `MessageBits`

This parameter contains the event number for which a callback function is to be registered. All system events also contain a copy of the device ID in bits 8 to 15.

`MaskBits`

This parameter contains a bitmask (to be applied to `MessageBits`) to specify the messages to which the callback applies. Setting this value to `0xffffffff` causes the callback to be made only on the exact event specified by `MessageBits`. Setting this value to `0` makes the callback on every event; setting the value to `0xff00` makes the callback on every event from a specific device.

`HandlerFunc:`

This parameter is a pointer to a handler function used to process the message. If this parameter is set to `NULL`, the function registered for that message is deregistered.
The function must be declared as follows:

```
int _cdecl Handler(MESSAGE *msg)
```

This function is called when the callback criteria is met. The function is always called when the application is in a state of blocking on `RimGetMessage`. The call is made in the registered application's context.
If the handler function returns `TRUE`, the message is passed to subsequent handler functions. If the handler function returns `FALSE`, the message is considered processed and is not forwarded. If the handler function modifies the message, then the modified message is subsequently forwarded to other handlers using `RimGetMessage`.
Multiple handler functions can process the same message. Re-registering a new function for the same event does not cancel the previous registration. If multiple handler functions examine the same message, the most recently registered handler function is called first. To cancel a previous registration, register the message again with `NULL` as the function pointer.

Return Value: This function returns `TRUE` if successful and `FALSE` if it is not successful. The function might be unsuccessful if all available memory is exhausted.

Remarks: This function is used to register a callback function to be called when certain message types are received. This utility enables certain functions to be called on certain events without having to check for the events on every call to `RimGetMessage`.

Note Registering callbacks is application specific. Registering callbacks for certain events does not affect the processing of messages in other applications; however, calls to `RimGetMessageRaw` can override the callback. Refer to `RimGetMessageRaw`.

RimReplyMessage

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

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

Parameters: `HTask`

The task handle of the application to which the message is to be sent

`Msg`

The pointer to the message to be returned to the task that called `RimSendSyncMessage`

Return Value: If the message is sent to a valid task and that task is waiting for a reply message from the caller, then `TRUE` is returned. `FALSE` is returned in all other cases.

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

Example: Refer to `GPS.C`.

RimRequestPowerOff

Posts a `POWER_OFF_REQUEST` message to the System Task

```
BOOL RimRequestPowerOff( void )
```

Return Value: This function returns `TRUE` if the power-off sequence was started. In this case, a subsequent call to `RimGetMessage` returns a `POWER_OFF` event.

If the serial port is in use by an application, the function returns `FALSE` without initiating the power-off sequence. This prevents events such as auto shut-off from spontaneously shutting down the radio modem while synchronizing or while connected to the simulator.

Remarks: If successful, this function posts a `POWER_OFF_REQUEST` message to the System Task, which then turns off the radio and sends a `POWER_OFF` notification to all application tasks.

RimSendMessage

Sends a message synchronously to another application

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

Parameters: `HTask`

The task handle of the application to which to send the message

`Msg`

A pointer to a `MESSAGE` structure

Return Value: If the message is sent to the destination task, this function returns `TRUE` after the destination task has received the message. If the destination task cannot receive a message immediately, 0 is returned. It is not guaranteed that the receiving task is the first or only task that is run before `RimSendMessage` returns.

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

Remarks: This function enables applications to send messages synchronously to other applications. This function does not return until the destination task has received the message. Tasks cannot synchronously send messages to themselves. Use the asynchronous `RimPostMessage` instead. Refer to “Operating system messaging” on page 65 for more information on asynchronous and synchronous sends.

RimSendSyncMessage

Sends a message to another application synchronously and waits for a reply

```
BOOL RimSendSyncMessage( TASK HTask, MESSAGE *Msg,
  MESSAGE *rReplyMsg )
```

Parameters:

`HTask`
: The task handle of the application to which the message is to be sent

`Msg`
: A pointer to the message to be sent

`ReplyMsg`
: A pointer to a message structure that is populated by the reply message from the receiving task

Return Value: If the message is sent to a valid task, and that task receives the message and calls `RimReplyMessage` appropriately, then `TRUE` is returned. If the task to which it is sent is invalid, or is terminated during the call to `RimSendSyncMessage`, then `FALSE` is returned and the value to which `ReplyMsg` points is undefined.

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

Remarks: The `RimSendSyncMessage` function enables an application to send messages to other applications synchronously and wait for a reply. This function does not return successfully until the message is received and the receiver completes a call to `RimReplyMessage` specifying the sender. This function call can cause applications to block indefinitely and should be used with caution if the sending application is also receiving messages.

See also: `RimReplyMessage`, `RimToggleMessageReceiving`

Example: Refer to `GPS.C`.

RimSetAlarmClock

Sets a timer alarm for an application

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

Parameters: `Time`

A pointer to a `TIME` structure

`Enable`

Specifies whether to enable the alarm clock

Return Value: `TRUE` if the alarm was set or `FALSE` if an error occurred

Remarks: This function sets a timer associated with the calling application. After the time has expired, the `ALARM_EXPIRED` event is sent to the task.

Each application can set its own alarm time.

RimSetDate

Sets the date on the real-time clock

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

Parameters: `Time`

A pointer to a `TIME` structure

Return Value: `TRUE` if the date is set or `FALSE` if the date is incorrectly formatted

Remarks: This function sets the date on the real-time clock

RimSetPID

Changes the attributes of the calling process

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

Parameters: `Pid`

A pointer to a `PID` structure

Return Value: No return value

Remarks: This function enables a process to change its own attributes. If the `Name` field is `NULL`, the field is not modified.

Example: Refer to `GPS.C`.

RimSetReceiveFromDevice

Selects from which devices an application receives messages

```
void RimSetReceiveFromDevice( DWORD Device,
  BOOL ReceiveFrom )
```

Parameters: `Device`

Defines the device from which to receive or stop receiving messages

`ReceiveFrom`

Specifies whether to start or stop receiving from the selected device

Return Value: No return value

Remarks: This function enables an application to select specifically from which devices it wants to receive messages. When message reception is turned off and on using `RimToggleMessageReceiving`, all settings made using `RimSetReceiveFromDevice` are restored.

RimSetTime

Sets the time on the real-time clock

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

Parameters: `Time`

A pointer to a `TIME` structure

Return Value: `TRUE` if the time is set or `FALSE` if time is incorrectly formatted

Remarks: This function sets the time on the radio modem real-time clock. The programmed time must be in 24-hour format

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

RimSetTimer

Sets a timer for an application

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

Parameters: `TimerID`

The identifier of the timer to be set

`Time`

The amount of time until the timer expires, and the time between subsequent timer expirations for periodic timers, specified in 1/100-second increments

`Type`

This parameter can be one of the following three values:

`TIMER_ONE_SHOT`	Timer expires once, in the amount of time specified by `Time`.
`TIMER_PERIODIC`	Timer expires for a period of time specified by `Time`.
`TIMER_ABSOLUTE`	Timer expires when the absolute tick count reaches the specified value. The absolute tick count can be obtained by calling `RimGetTicks.`

Return Value: `TRUE` if the timer was set successfully, or `FALSE` otherwise

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

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

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

Example: Refer to `GPS.C`.

RimSleep

Stops execution of an application for a specified period of time

```
BOOL RimSleep( DWORD Ticks )
```

Parameters: `Ticks`
: Defines the number of 10-millisecond periods for which the task is to sleep

Return Value: `TRUE` if the application is stopped successfully, `FALSE` otherwise

Remarks: The `RimSleep` function enables an application to stop running for a specified period of time. Messages do not wake the application, so you must ensure that message queues do not overflow.

See also: `RimToggleMessageReceiving`

RimSprintf

Formats text into a buffer

```
int RimSprintf( char *Str, int Maxlen, const
  char *Fmt, … )
```

Parameters: `Str`
: A block of memory into which the formatted string is to be placed

`Maxlen`

The maximum number of characters that can be placed into the buffer

`Fmt`

The format string used to determine the format of the data

`...`

A list of parameters following the format string to supply information to the format string

Return Value: This function has a return value of the number of characters placed into the memory pointed to by `Buffer`, excluding the terminating `NULL` character. The function returns `–1` if there is insufficient space, in which case `Maxlen` characters have been placed into the buffer.

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

Remarks: This function is similar to the standard C function `sprintf`, but with a buffer size parameter added. Field specifications appear in the following form:

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

(Options are indicated by the [ ]. Do not include the [ ].)

`-`	Left-justify the field; it is right-justified otherwise
`0`	Pad with ‘0’ to the width of the field, if it is right-justified; the default is to use spaces
`width`	The minimum width of the formatted data for this field
`precision`	For numbers, the minimum number of digits to include; for strings, the maximum number of characters to include
`l`	The number is to be formatted as a long integer value

`format`	Data format type; one of the following options: `d` - integer in decimal format `i` - integer in decimal format `b` - integer in binary format `o` - integer in octal format `u` - unsigned integer in decimal format `x` - integer in hexadecimal format, lowercase letters `X` - integer in hexadecimal format, uppercase letters `s` - pointer to a string; if a `precision` is not specified, the string must be `NULL`-terminated `c` - character

Example: Refer to `GPS.C.`

RimStackUsage

Helps choose an appropriate value for the task stack size

```
DWORD RimStackUsage( void )
```

Return Value: This function returns the maximum stack size used by the current task since the previous call to `RimStackUsage`. This number includes the stack space required by `RimStackUsage` itself.

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

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

RimTaskYield

Yields execution to other tasks

```
void RimTaskYield( void )
```

Return Value: No return value

Remarks: This function allows an application to yield control, allowing other applications to run. Applications yield control during intensive CPU operations so that other applications can run. If there are no other applications with messages pending, `RimTaskYield` returns immediately.

Example: Refer to `PINGOEM.CPP`, `GPS.C` and `GPSDEMO.C`.

RimTerminateThread

Terminates the calling thread

```
void RimTerminateThread( void )
```

Return Value: No return value

Remarks: This function enables a thread to terminate itself. This function does not return. Returning from the main function of a thread has the same effect as calling `RimTerminateThread`.

RimToggleMessageReceiving

Toggles rejection of incoming messages

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

Parameters: `ReceiveMessages`

  Specifies whether the application is going to receive messages or turn off message reception

Return Value: No return value

Remarks: The `RimToggleMessageReceiving` function enables an application to reject all incoming messages, or to accept them again after having rejected them for a period of time.

This function does not change which messages are sent to the application; this is done using the function `RimSetReceiveFromDevice`. Any attempt to send a message to an application that has turned off message receiving fails and the sending call returns `FALSE`.

See also: `RimSetReceiveFromDevice`

RimVsprintf

Formats text into a buffer

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

Parameters: `Buf`

A block of memory into which the formatted string is to be placed

`Maxsize`

The maximum number of characters to be placed into the buffer

`Fmt`

The format string used to determine the format of the data

`Argp`

A pointer to a variable parameter list specifying the data to be formatted

Remarks: This function behaves similarly to the standard C library function `vsprintf` and identically to `RimSprintf`. Refer to `RimSprintf` for information on how to use the function.

RimWaitForSpecificMessage

Waits to receive a specific message

```
TASK RimWaitForSpecificMessage( MESSAGE *Msg,
      MESSAGE *Compare, DWORD Mask )
```

Parameters: `Msg`

A pointer to the `MESSAGE` structure that is populated with the received message

`Compare`

A pointer to a `MESSAGE` structure that is compared with incoming messages

`Mask`

This parameter is a bitmask comprised of any of `MC_DEVICE`, `MC_EVENT`, or `MC_SUBMSG`. This mask determines which parts of the message structure to compare.

Return Value: If the Mask is valid, this function returns the task ID of the specific message for which the application is waiting.

If the Mask is invalid, this function returns `FALSE`.

Remarks: `RimWaitForSpecificMessage` enables an application to receive a specific message at a critical time. All other messages sent to the calling application are queued until `RimGetMessage` is called. When using this function call, you should ensure that the message queue does not overflow.

See also: `RimSetReceiveFromDevice`

9
Serial Communications API

This chapter provides information on Serial API functions, constants, and error codes.

The Serial Communications API enables applications to access the external serial ports on the radio modem. All low-level hardware functionality is built into the operating system, relieving you from programming complex serial functions.

Serial API functions

SerialClose

Closes the specified serial port, disables line drivers, frees any buffer memory that was allocated when the specified serial port was opened

```
void SerialClose( unsigned port )
```

Parameters: `port`
The serial port that is being used by the routine

Return Value: No return value

Remarks: This function closes the specified serial port, disables the line drivers, and frees any buffer memory that was allocated when the specified serial port was opened. `SerialClose` is called by the system automatically when the radio modem is shut off. Applications must not assume that the specified serial port is still open when the radio modem starts again.

Example: Refer to `SERIALDEMO.C` and `GPSDEMO.C`.

SerialGetDCD

Queries the state of the data carrier detect (DCD) line on the specified serial port.

```
BOOL SerialGetDCD( unsigned port )
```

Parameters: `port`
The serial port that is being used by the routine

Return Value: The current state of the DCD line on the specified serial port

Remarks: The DCD line is actually the DCD control line as driven by an attached computer. On the device, the port must equal 0, as the hardware lines for UART 1 are not brought out to the connector.

SerialGetDTR

Reads the state of the host data terminal ready (DTR) modem control line on the specified serial port

```
BOOL SerialGetDTR( unsigned port )
```

Parameters: `port`
The serial port that is being used by the routine

Return Value: The current state of the DTR control line on the specified serial port.

Remarks: The DTR line is actually the DTR control line as driven by an attached computer. On the device, the port must equal 0, as the hardware lines for UART 1 are not brought out to the connector.

SerialGetRTS

Reads the state of the host request to send (RTS) modem control line on the specified serial port

```
BOOL SerialGetRTS( unsigned port )
```

Parameters: `port`
The serial port that is being used by the routine

Return Value: The current state of the RTS control line on the specified serial port

Remarks: The RTS line is actually the RTS control line as driven by an attached computer. On the device, the port must equal 0, as the hardware lines for UART 1 are not brought out to the connector.

SerialOpen

Allocates serial and transmission FIFO buffers in memory, enables the serial driver and hardware serial line drivers for the specified serial port

```
BOOL SerialOpen( unsigned port, const struct
 SerialOpenParms *parms, DWORD SizeofParms)
```

Parameters: `port`
The serial port that is being used by the routine

`parms`

The parameter provides a structure, detailed below, that contains all relevant serial port information. Setting this argument to `NULL` defaults the specified port to 8 data bits, no parity, and 1 stop bit.

```
struct SerialOpenParms{
    DWORD baud;
    int   databits;
    int   parity;
    int   stopbits;
    DWORD RxBufferSize;
    DWORD TxBufferSize;
    enum FlowControlType flow;
};
```

Field	Description
`baud`	The baud rate that the serial port is to use
`databits`	The number of bits to use (7 or 8)
`parity`	The parity method to use, one of: `SERIAL_NO_PARITY` (0) `SERIAL_EVEN_PARITY` (1) `SERIAL_ODD_PARITY` (2)
`stopbits`	The number of stop bits to use (1 or 2)
`RxBufferSize`	The receive FIFO serial buffer size; it must be a power of two; values that are not a power of two are rounded up to a power of two
`TxBufferSize`	The transmit FIFO serial buffer size; it must be a power of two; values that are not a power of two are rounded up to a power of two
`flow`	The type of flow that occurs through the specified serial port

`SizeofParms`

The size of the `parms` struct

Return Value: `TRUE` if successful and `FALSE` if unsuccessful

Remarks: This function allocates serial and transmission FIFO buffers in memory, and enables the serial driver, as well as hardware serial line drivers. It is advisable to always close the specified serial port when it is not in use, as the hardware line drivers consume extra power while the port is open. If port 0 is opened, then this function should be called on startup immediately to transfer control of this port from the radio to the user. If this function is not called immediately, then the radio takes control of the port, and the user is unable to control it.

Example: Refer to `SERIALDEMO.C` and `GPSDEMO.C`

SerialRead

Reads data received over the specified serial port

```
int SerialRead(unsigned port, BYTE *data, int
  length)
```

Parameters:

`port`
: The serial port that is being used by the routine

`data`
: This parameter is a pointer to a buffer for holding the received data. Set this to `NULL` if the data is to be discarded. This can be used to flush the receive buffer.

`length`
: The maximum number of bytes to remove from the receive buffer

Return Value: This function returns the number of bytes actually read; this will be the number of bytes in the serial receive FIFO when the function is called. `SerialRead` returns only the bytes received when `SerialRead` was called, and does not wait for extra bytes to arrive. If `SerialRead` returns less than what was passed in as `length`, the serial receive buffer is empty after the call. Emptying the buffer ensures that a new `SERIAL_RX_AVAILABLE` message can be generated after new bytes arrive.

If `data` is `NULL`, `SerialRead` empties the receive buffer and returns 0.

Example: Refer to `GPSDEMO.C.`

SerialReadChar

Reads one byte from the specified serial port

```
int SerialReadChar( unsigned port )
```

Parameters: `port`

The serial port that is being used by the routine

Return Value: This function returns the byte read from the serial port. It returns a negative value if an error occurs or no bytes are available.

Remarks: This function is equivalent to calling `SerialRead` with a length of one.

Example: Refer to `SERIALDEMO.C` and `GPSDEMO.C`.

SerialRxCount

Returns the number of characters that have been received in the serial receive FIFO

```
int SerialRxCount( unsigned port )
```

Parameters: `port`

The serial port that is being used by the routine

Return Value: The number of characters that have been received in the buffer

Remarks: This function returns the number of characters that have been received in the serial transmit FIFO. It returns 0 if nothing has been received.

SerialRxFlush

Flushes the Receive buffer of the specified port

```
void (SerialRxFlush( unsigned port)
```

Parameters: `port`

The serial port that is being used by the routine

Return Value: None

SerialSend

Places the bytes to be sent into the transmission serial FIFO

```
int SerialSend( unsigned port, BYTE * *data,
  int length )
```

Parameters:
`port`
: The serial port that is being used by the routine

`data`
: A pointer to the data to be sent

`length`
: The number of bytes to be sent

Return Value: This function returns the number of bytes that are placed in the transmission FIFO. It returns a negative value if an error occurs. `SerialSend` places the data in the transmission FIFO and starts transmitting, but returns before the data leaves the port.

Remarks: This function places the bytes to be sent into the transmit serial FIFO. The function returns immediately and indicates how many bytes are placed in the buffer. If items larger than the serial buffer are to be sent, the item must be split into several chunks, while waiting for the `SERIAL_TX_EMPTY` message.

Example: Refer to `SERIALDEMO.C, GPSDEMO.C.`

SerialSendChar

Returns the number of bytes that can be placed in the transmission (first-in, first-out)

```
int SerialSendChar( unsigned port, BYTE character )
```

Parameters:
`port`
: The serial port that is being used by the routine

`character`
: The single character to be sent

Return Value: This function returns the number of bytes that are placed in the transmit FIFO. It returns a negative value if an error occurs.

Remarks: This function is equivalent to calling `SerialSend` with only one byte.

Example: Refer to `SERIALDEMO.C.`

SerialSetCTS

Sets clear to send (CTS) modem control line

```
void SerialSetCTS( unsigned port, BOOL state )
```

Parameters: `port`
The serial port that is being used by the routine

`state`
The desired state for the CTS signal (0 or 1)

Return Value: No return value

Remarks: The clear to send (CTS) signal is presented as data set ready (DSR) to an attached computer. On the device, the port must equal 0, as the hardware lines for UART 1 are not brought out to the connector. Calling this function to set the CTS state, when hardware flow control has been selected, generates an error.

SerialSetDSR

Sets data set ready (DSR) modem control line

```
void SerialSetDSR( unsigned port, BOOL state )
```

Parameters: `port`
The serial port that is being used by the routine

`state`
The desired state for the DSR signal (0 or 1)

Return Value: No return value

Remarks: The DSR signal is presented as DSR to an attached computer.

Example: Refer to `SERIALDEMO.C.`

SerialSetFlowControl

Controls data flow through the specified serial port

```
void SetFlowControl ( unsigned port, enum
  FlowControlType flow )
```

Parameters: `port`

The serial port that is being used by the routine

`flow`

This parameter describes the type of flow control at the specified serial port.

```
typedef enum FlowControlSettings {
    SERIAL_NO_FLOW_CONTROL,
    SERIAL_SOFTWARE_FLOW_CONTROL,
    SERIAL_RTS_CTS_FLOW_CONTROL,
} FlowControlType;
```

Return Value: No return value

Remarks: This routine is used to modify the type of data flow control at the specified serial port. It enables software flow control, RTS/CTS flow control, or no data flow.

SerialSetRI

Sets ring indicate (RI) state

```
void SerialSetRI( unsigned port, BOOL state )
```

Parameters: `port`

The serial port that is being used by the routine

`state`

The desired state for the RI signal (0 or 1)

Return Value: No return value.

Remarks: The RI signal is presented as RI to an attached computer

SerialSettings

Changes the configuration of the specified serial port

```
void SerialSettings ( unsigned port, DWORD baud,
  int databits, int parity, int stopbits )
```

Parameters: `port`

The serial port that is being used by the routine

`baud`

The baud rate that the specified serial port is to use

`databits`

The number of bits to use (7 or 8)

`parity`
: The parity method to use: `SERIAL_NO_PARITY` (`0`), `SERIAL_EVEN_PARITY` (`1`), or `SERIAL_ODD_PARITY` (`2`)

`stopbits`
: The number of stop bits to use (1 or 2)

Return Value: No return value

Remarks: This function changes the configuration of the specified serial port

Example: Refer to `SERIALDEMO.C.`

SerialStandby

Puts the specified serial port in standby mode

```
void SerialStandby( unsigned port, BOOL state )
```

Parameters: `port`
The serial port that is being used by the routine

`state`
The desired state (1 for standby on, 0 for standby off)

Return Value: No return value

Remarks: This function sets the standby state of the specified serial port

Example: Refer to `SERIALDEMO.C`.

SerialTxCount

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

```
int SerialTxCount( unsigned port )
```

Parameters: `port`
The serial port that is being used by the routine

Return Value: The number of characters in the transmit buffer

Remarks: This function returns the number of characters that remain to be transmitted in the serial transmit FIFO. It returns 0 if the transmit FIFO and hardware are completely empty.

Example: Refer to `GPSDEMO.C`.

SerialTxFlush

Flushes the transmit buffer of specified port

```
void SerialTxFlush( unsigned port )
```

Parameters: `port`
The serial port that is being used by the routine

Return Value: No return value

Serial API error codes

Serial Communications API functions return the following error codes:

Error Code	Meaning
`SERIAL_ERROR_NOT_OPEN`	Indicates that the serial port has not been opened
`SERIAL_ERROR_PARITY`	Indicates that a receive error has occurred
`SERIAL_ERROR_FRAMING`	Indicates that a frame error has occurred
`SERIAL_ERROR_OVERRUN`	Indicates an error in the serial buffer; some data was lost because received data could not be read quickly enough
`SERIAL_ERROR_BREAK`	Indicates that a break condition was detected (occurs when the receive data line is held at a logic zero state for longer than the time required to receive a byte and a stop bit)

10
Radio API

This chapter provides information on Radio API structures, functions, and error codes.

The Radio API provides access to the Mobitex network using simple API function calls to send and receive data. You do not need extensive knowledge of the Mobitex network to use these function calls.

Radio events are announced to applications through the message system and provide information on the status of incoming and outgoing packet communications.

Radio API structures

The Radio API uses the following structures.

MPAK_HEADER structure

This structure represents the header data in an Mobitex packet (MPAK).

```
typedef struct {
    long Sender;
    long Destination;
    int MpakType;
    int HPID;
    int Flags;
    TIME Timestamp;
    long lTime;
    int TrafficState;
} MPAK_HEADER;
```

Field	Description
Sender	The sender's MAN number; filled in automatically when sending MPAKs
Destination	The addressee's MAN number or the destination to which the MPAK is to be sent
MpakType	Type of MPAK. This can be one of the following: • MPAK_TEXT • MPAK_DATA • MPAK_STATUS • MPAK_HPDATA For all other MPAK types, this value is set to 0.
HPID	HPID value for HHPDATA MPAK types
Flags	Logical OR of the flag values for the MPAK; this can be a combination of FLAG_MAILBOX and FLAG_POSACK.
Timestamp	Parsed Mobitex timestamp.
lTime	MPAK raw Mobitex timestamp, a minute counter since the start of 1985
TrafficState	MPAK traffic state. This can be one of the following: • TS_MESSAGE_OK • TS_MESSAGE_FROM_MAILBOX • TS_MESSAGE_IN_MAILBOX • TS_CANNOT_BE_REACHED • TS_ILLEGAL_MESSAGE • TS_NETWORK_CONGESTED • TS_TECHNICAL_ERORR • TS_DESTINATION_BUSY

RADIO_INFO structure

This structure stores general information about the state of the radio.

```
typedef struct {
    int   RadioOn;
    DWORD LocalMAN;
    DWORD ESN;
    int   Base;
    int   Area;
    int   RSSI;
    WORD  NetworkID;
    DWORD FaultBits;
    BOOL  Active;
    BOOL  PowerSaveMode;
    BOOL  LiveState;
    BOOL  TransmitterEnabled;
} RADIO_INFO;
```

Field	Description
`RadioOn`	One of `RADIO_ON` or `RADIO_OFF`
`LocalMAN`	The modem’s MAN number, as a 32-bit value
`ESN`	The electronic serial number, as a 32-bit value
`Base, Area`	Current base and Area ID where the modem is located or where the modem was last in network coverage; the Base and Area together uniquely identify a base station in the Mobitex network
`RSSI`	RSSI value in the range of –113 to –40, or `RSSI_NO_COVERAGE;` values above –90 are generally reliable network coverage
`NetworkID`	Network ID of the network on which the radio modem is located. (This value is 0xB433 in the U.S.)
`FaultBits`	Flags indicating various problems with the modem.

Field	Description
`PowerSaveMode`	Status of the low-power mode; always 1 on the radio modem: 1 = power save mode 0 = express
`LiveState`	Status of the live state mode 1 = LIVE state, 0 = DIE state
`TransmitterEnabled`	Status of the transmitter mode 1 = Tx enabled, 0 = Tx disabled

SKIM_INFO structure

This structure is used to query the radio about R14N `skipnum` settings.

```
typedef struct {
    BYTE  SkipNum;
    BYTE  ProtocolRevision;
    BYTE  SkipTrans;
    BYTE  Mode;
} SKIPNUM_INFO;
```

Field	Description
`SkipNum`	Current `Skipnum` value used by the modem
`ProtocolRevision`	Mobitex protocol revision (0 for pre-R14N)
`SkipTrans`	`SkipTrans` value of the Mobitex network
`Mode`	Status of the low-power mode. 1 = low-power mode, 0 = express mode

NETWORKS_INFO structure

This structure is used to query the radio for supported network IDs.

```
typedef struct {
    int   DefaultNetworkIndex;
    int   CurrentNetworkIndex;
    int   NumValidNetworks;
    struct {
       WORD NetworkId;
       BYTE NetworkName[10];
    } Networks[10];
} NETWORKS_INFO;
```

Field	**Description**
`DefaultNetworkIndex`	Default network
`CurrentNetworkIndex`	Current network
`NumValidNetworks`	Number of valid networks
`NetworkId`	Network frame synchronization word (0xB433 in the U.S.) Part of the `Networks` substructure in `NETWORKS_INFO`
`NetworkName[10]`	Name of the network Part of the `Networks` substructure in `NETWORKS_INFO`

Radio API functions

RadioAccelerateRetries

Causes the radio to retry transmitting more aggressively

```
void RadioAccelerateRetries( int mpakTag )
```

Parameters: `mpakTag`

The tag of the MPAK in the radio to accelerate (not currently used)

Remarks: When the radio has difficulty transmitting an MPAK to the base station due to network congestion or poor network coverage, it normally increases the interval between transmission retries to allow conditions to improve. `RadioAccelerateRetries` causes the radio to retry sending the MPAK in the modem more aggressively. This decreases battery life in exchange for stronger attempts to send the MPAK. `RadioAccelerateRetries` should normally only be called based on user action that indicates that the user is waiting for a packet to be sent (such as the user selecting Resend for data that has already been submitted by an application).

RadioCancelSendMpak

Cancels a submitted MPAK

```
int RadioCancelSendMpak( int mpakTag )
```

Parameters: `mpakTag`

This parameter is the tag assigned by the application server when the packet is submitted to the radio modem for transmission. A value of -1 causes all MPAKs queued for transmission by the calling application to be cancelled.

Return Value: This function returns the number of MPAKs that were cancelled. It returns a negative value if an error occurs.

Remarks: This function attempts to cancel a submitted packet identified by the tag number. If this function is called before the MPAK is transmitted, the MPAK is returned to the application as cancelled, provided that it has not already been sent. There is no guarantee, however, that a cancelled MPAK was not already received by the Mobitex network.

RadioChangeNetworks

Changes the current radio network (Rogers AT&T in Canada, Cingular Interactive in the United States)

```
void RadioChangeNetworks( DWORD NetworkId,
  BYTE * NetworkName )
```

Parameters `NetworkId`

The new network’s ID number.

`NetworkName`

Name of the network to which the current network is being changed.

Return Value: No return value

Remarks: This function changes the current network to the specified network. This could be necessary if the application requires access to networks in both Canada and the United States.

RadioDeregister

De-registers applications from receiving radio events

`void RadioDeregister( void )`

Return Value: No return value

Remarks: This function de-registers the current application so that it no longer receives `RADIO` events. Any MPAKs that the de-registering application has pending for transmission are cancelled and returned to the application. Thus, it is still possible for the application to receive some radio events after de-registering.

RadioGetAvailableNetworks

Programs the available networks into the radio modem

`void RadioGetAvailableNetworks( NETWORKS_INFO *info)`

Parameters: `info`

Pointer to a NETWORKS_INFO structure (refer to “Radio API structures” on page 113).

Return Value: No return value.

Remarks: Enables you to query the radio modem and determine which networks have been programmed in.

RadioGetDetailedInfo

Retrieves the current state of the radio

`void RadioGetDetailedInfo( RADIO_INFO *info )`

Return Value: No return value

Remarks: Retrieves the current state of the radio, such as MAN number, RSSI, on / off, powersave / express, base, and area, into a `RADIO_INFO` structure (refer to “Radio API structures” on page 113 for details)

RadioGetMpak

Retrieves the data of a received MPAK

```
DWORD RadioGetMpak ( int mpakTag, MPAK_HEADER
  * header, BYTE * data )
```

Parameters: `mpakTag`

This parameter is the `MPAK_TAG` value from the `MESSAGE_RECEIVED` message. The `MPAK_TAG` value has a limited life span. For received MPAKs, the tag must be used before getting the next message or yielding.

`header`

This parameter is a pointer to an `MPAK_HEADER` structure (refer to “Radio API structures” on page 113 for details). The information extracted from the MPAK header is placed in this structure.

If this pointer is `NULL`, no header is extracted, and the raw MPAK is placed in the buffer.

`data`

This parameter is a pointer to a buffer large enough to contain the MPAK. The amount of space required can be determined by calling `RadioGetMpak`. It is recommended that this parameter always point to a buffer of at least 512 bytes. If the header pointer is `NULL`, the raw MPAK is placed in the buffer. In this case, the buffer should be at least 560 bytes.

If the data pointer is `NULL`, the MPAK is not copied.

Return Value: If `header` is not `NULL`, the number of data bytes in the data portion of the MPAK (0 to 512) is returned if successful. If `header` is `NULL`, the length of the entire MPAK is returned.

The function has a return value of -1 if it is unsuccessful.

Remarks: When an MPAK is received, the `MPAK_TAG` value is contained in the message. This tag value is used to obtain subsequent information about the MPAK.

This function can also be used to get copies of MPAKs that are queued for transmission. MPAKs that are queued for transmission can be recalled at any time until `RimTaskYield` or `RimGetMessage` are called.

`RadioGetMpak` can be used in several ways:

- You can obtain only the header, by setting the data pointer to `NULL`.
- You can obtain both the header and the MPAK. Both pointers point to their respective data areas. Only the data portion of the MPAK is copied into the data buffer.
- You can obtain the raw MPAK. The header pointer is set to `NULL`, while the data pointer points to a buffer for the raw MPAK. The entire MPAK, including header information, is copied into the data buffer.

The MPAK is only guaranteed to be available until the application yields control to the system (via `RimGetMessage` or `RimYield`). The MPAK remains available until all applications that have registered to receive MPAKs have received the `RADIO MESSAGE_RECEIVED` message. After all registered applications have received this message, the MPAK is released the next time control is yielded to the system (through `RimGetMessage` or `RimTaskYield`).

RadioGetSignalLevel

Gets the current signal strength

```
int RadioGetSignalLevel( void )
```

Return Value: Radio signal level in dBm, if the modem is in an area of wireless network coverage; the value is typically between -121 dBm and -40 dBm

If the modem is out of network coverage, the return value is -256 (`RSSI_NO_COVERAGE`) or less.

Remarks: The return value is always negative. A higher number (closer to 0) indicates greater strength of the received signal. For example, –90 dBm. indicates greater coverage than -93 dBm.

Example:

```
// Displays strength of received radio signal
int level = RadioGetSignalLevel();

if (level > RSSI_NO_COVERAGE){
  sprintf( buffer, “Level = %d dBm”, level );
} else {
  sprintf( buffer, “No coverage”);
```

}*RadioOnOff*

Checks / changes radio status (on / off)

`int RadioOnOff( int mode )`

Parameters: `mode`

Specifies the new state of the radio; the `mode` parameter can be one of the following values:

`radio_on`	Turns on the radio
`radio_off`	Turns off the radio
`radio_get_onoff`	Returns the current on / off state

Return Value: The function returns the state of the radio before `RadioOnOff` was called, and can be one of several values

`radio_on`	The radio is on
`radio_off`	The radio is off, or turning off
`radio_lowbatt`	The radio is on but the battery is too low for it to be operational

Remarks: This function enables the applications to check and modify the on / off state of the radio. The radio must be explicitly turned on if applications want to use it, as its default state is off if any applications are loaded.

Note Refer to `RadioGetDetailedInfo` to check other details of the radio’s state.

RadioRegister

Registers applications for radio events

```
void RadioRegister( void )
```

Return Value: No return value

Remarks: Applications must call this function to receive notification of `RADIO` events (including received MPAKs). Applications that have not registered for radio events cannot send or receive MPAKs. After calling `RadioRegister`, the application receives a `SIGNAL_LEVEL` message if the radio is on or receives a `RADIO_TURNED_OFF` message if the radio is off.

RadioRequestSkipnum

Sets and requests the `skipnum` parameters used in R14N networks

```
int RadioRequestSkipnum( SKIPNUM_INFO * SkipInfo,
  int Skipnum )
```

Parameters: `SkipInfo`

This parameter is a pointer to a `SKIPNUM_INFO` structure (refer to “Radio API structures” on page 113 for details). This structure is filled with the current parameters of the R14N skipping algorithm. These parameters reflect the value of `Skipnum` that was used at the time that `RadioRequestSkipnum` was called.

If `Info` is a `NULL` pointer, the structure will not be filled in.

`Skipnum`

This parameter represents the new value of `Skipnum` to use. Legal values are 1, 2, 4, 8, and 16. A value of 0 indicates an information request without a change request.

Return Value: This function returns the value of `Skipnum` that was used before `RadioRequestSkipnum` was called. It returns negative if an error occurs.

Remarks: This function is used to set and request the SVP skip parameters used in networks supporting the R14N level of firmware or higher, such as the Cingular Interactive network in the United States.

The `skipnum` value is how many 10-second periods the modem waits before turning on to see if the network has traffic to address to it. Thus, an additional delay of up to 10 * `skipnum` seconds can be introduced on unsolicited traffic from the base station. `Skipnum` does not affect timing of traffic going into the network, nor does it affect traffic coming from the network within a minute of sending into the network.

When the `skipnum` value changes, the modem transmits to the network which `skipnum` interval is used. A `skipnum` value of 1 provides the fastest delivery of unsolicited traffic from the network, while larger values of `skipnum` save battery power since the receiver is not on as often. `Skipnum` values of 1, 2, or 4 are recommended, while 4 is the default. The gain in battery life of settings greater than 4 is small, so values above 4 are not recommended.

Refer to Ericsson's documentation on R14N for more details on `skipnum` and how it affects various aspects of the Mobitex protocol.

RadioResumeReception

Indicates that the application is ready to receive MPAKs again.

```
void RadioResumeReception ( void )
```

Return Value: No return value

Remarks: This function is used to indicate that the application is ready to receive MPAKs again after `RadioStopReception` is called. If `RadioStopReception` is used to save an MPAK, the MPAKis again received with a `MESSAGE_RECEIVED` message, as if it had just been received by the radio.

This function must be called by the same task or thread that calls `RadioStopReception`. Each task that calls `RadioStopReception` must call this function before more MPAKs can be received.

RadioRoamNow

Starts the modem searching for the best base station

```
void RadioRoamNow( void )
```

Return Value: No return value

Remarks: Putting the radio modem into roaming mode causes the radio modem to continually search for the best base station, to optimize mobile reception.

RadioSendMpak

Submits an MPAK for transmission by the radio

```
int RadioSendMpak( MPAK_HEADER * header, BYTE * data,
int length)
```

Parameters: `header`

This parameter is a pointer to an `MPAK_HEADER` structure (refer to “Radio API structures” on page 113 for details). This structure contains information for building the MPAK header, including the type of MPAK and the addressee. If the application wants to build the MPAK and header itself, the `header` parameter is set to `NULL`, and no header is added to the MPAK.

Note	The Sender field of the header is always filled by the application server to the modem’s MAN, and does not need to be set by the application.

`data`

This parameter is a pointer to a buffer containing the data bytes to be included in the MPAK.

Return Value: A tag is assigned to the MPAK by the application server. If the sequence identification is negative, the message cannot be queued for sending. The returned tag value is always less than `MAX_QUEUED_MPAKS`, which is currently defined as 7.

Remarks: RadioSendMpak submits an MPAK for transmission by the radio. If an MPAK has already been submitted for transmission by this or any other application, the MPAK is queued. If more than 4 MPAKs are already queued, RadioSendMpak fails and returns a negative error code.

RadioSendMpak copies the data provided. The data that is pointed to when the call is made can be deleted after the call returns.

Example:

```
// Send an Hpdata 123 MPAK with data ‘Hello’ to MAN
// 123456. Send a Status 10 MPAK to MAN 123456.
{
  MPAK_HEADER header;
  int temp;

  header.Destination = 123456;
  header.MpakType = MPAK_HPDATA;
  header.Flags = FLAG_POSACK;
  header.TrafficState = TS_MESSAGE_OK;
  header.HPID = 123;

  tag1 = RadioSendMpak(header, “Hello”, 5);
  // Send a status MPAK. Header is mostly set up
  // already.
  header.MpakType = MPAK_STATUS;
  temp = 10;
  tag2 = RadioSendMpak(header, &temp, 1);
}
```

RadioStopReception

Indicates that the radio is not ready to receive MPAKs

`void RadioStopReception ( int mpakTag )`

Parameters: mpakTag

If RadioStopReception is called in response to a MESSAGE_RECEIVED message, the mpakTag value can be passed into the function as a parameter. After RadioResumeReception is called, the saved MPAK is resent to the calling application.

Remarks: The `RadioStopReception` function stops the radio modem from receiving MPAKs. It is intended for use when all buffers for receiving MPAKs are full. This function should be used only if no more memory can be allocated to save received data. `RadioStopReception` causes the radio to eventually stop receiving MPAKs for all applications running on the radio modem.

Further MPAKs can still be received after `RadioStopReception` is called, as they might already be in the calling task's message queue. These MPAKs can still be saved by calling `RadioStopReception` again

Radio API error codes

The following error codes pertain to radio function return values.

Error Codes	Meaning
`RADIO_APP_NOT_REGISTERED`	Applications must be registered for `RADIO` events to be allowed to send MPAKs. Attempting to send MPAKs without being radio-registered returns this error code.
`RADIO_MPAK_NOT_FOUND`	Attempting to fetch an MPAK with a tag value that has expired produces this error code. MPAKs must be fetched before the task yields control to other tasks.
`RADIO_NO_FREE_BUFFERS`	Attempting to send an MPAK with all the radio's outgoing buffers full produces this error code.
`RADIO_BAD_DATA`	Attempting to send an MPAK with format data that cannot be used in an MPAK produces this error code.
`RADIO_BAD_TAG`	Attempting to fetch an MPAK with a tag value outside the legal range produces this error code.
`RADIO_ERROR_GENERAL`	This is a generic radio error.
`RADIO_ILLEGAL_SKIPNUM`	Attempting to set the Skipnum value with `RimRequestSkipnum` to any value other than 1, 2, 4, 8, or 16 produces this error.

11
File System API

This chapter provides information on File System functions and error codes.

The File System API enables applications to access the radio modem’s persistent memory.

The File System API provides sufficient capabilities for most applications. If your application requires additional functionality, you can use the Database API. Refer to the *Database API Reference* for more information.

File system API functions

DbAddOrphan

Adds an orphan record to a database

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

Parameters: db

A database handle

orphan

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

Return Value: One of the following error codes is returned:

```
DB_OK
DB_ERR_BAD_HANDLE
DB_ERR_NOT_DIR
DB_ERR_NOT_ORPHAN
```

Remarks: This function is atomic, that is, the record is either appended to the database or the database is unchanged. No intermediate state exists. This enables data consistency to be preserved if the system crashes for any reason during the function call.

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

DbAddRec

Creates a new record (database or orphan)

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

Parameters: db

This parameter is the handle to the database to which the record is appended. Any negative value indicates that an orphan record should be created. An orphan record is similar to any other record, except that it is not included in a database. It can be made part of a database at a later time.

`size`

This parameter is the size of the record, in bytes. It must be between 0 and 65534 (2^{16} - 2), inclusive. At any particular instant, the maximum possible record size can be limited due to flash memory fragmentation. Refer to `DbMaxNewRecSize`.

`data`

This parameter is the pointer to the data that is written into the new record. If it is `NULL`, the data is assumed to have all bits set to 1. This can be convenient when used in conjunction with the `DbAndRec` function.

Return Value: This function has a return value of a non-negative handle to the newly created record. One of the following negative error codes can also be returned:

```
DB_ERR_BAD_SIZE
DB_ERR_NO_HANDLE
DB_ERR_NO_SPACE
DB_ERR_BAD_HANDLE
DB_ERR_NOT_DIR
```

Warning! The record handle returned by this function is intended for temporary database identification only and persists for the lifetime of a database on the particular system. Storing record handles (or database handles) within the permanent data is not advised since the data is not portable. It cannot be meaningfully copied to another system.

Remarks: This function is atomic, that is, the record is either completely written and made part of the database or the database is unchanged. No intermediate state is visible. This enables data consistency to be preserved if the system stops responding for any reason during the function call.

It is valid for `data` to point to portions of the file system (that is, through the memory-mapped file access mechanism). The entire operation is slightly slower, however. To write to the flash memory, the memory must be configured to prevent reading it simultaneously. As a result, the function must copy from flash memory to flash memory through RAM.

An orphan is a database record that has not yet been assigned to any database. An orphan record is useful for allocating temporary data in the file system. You can also use orphan records to create a record and then add it to the database as a separate operation.

Note Orphan database records are regular database records except that they are not associated with any file. Orphan records are automatically deleted after a device resets, as it is not possible to get the handle of an orphaned record.

DbAndRec

ANDs data to an existing record

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

Parameters: `rec`

The target record

`mask`

This parameter is a pointer to the data mask to be AND'ed to the current contents of the record. If it is `NULL`, the data mask is assumed to have all bits cleared to 0.

`size`

The data mask size in bytes

`offset`

The offset of the area to which to be AND'ed, from the beginning of record

Note As a precondition, `offset` + `size` must be less than or equal to the record size.

Return Value: One of the following error codes is returned:

`DB_OK`
`DB_ERR_BOUNDS`
`DB_ERR_BAD_HANDLE`
`DB_ERR_IS_DIR`

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

This function is not atomic. If the system stops responding during the function call, the record can be left in a partially updated state. The update order is not specified. Some parts might be completely updated, other parts might contain old data, and some parts might contain an arbitrary mix of old and new data. Some bits might be marginally updated, with charge values between 1 and 0. Such bits can return varied results on successive read operations. You must ensure that the data is correctly interpreted.

`DbAndRec` performs record modification in place. This uses less RAM and is substantially faster than the alternative of copying the record, modifying the copy, and calling `DbReplaceRec` to make the changes permanent. But, `DbAndRec` is not atomic and changes are not verified completely. Therefore, `DbAndRec` should be used only with non-critical transient data.

DbDelete

Deletes a database

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

Parameters: `fileName`

Pointer to the name of the database to be deleted from the file system

Return Value: One of the following error codes is returned:

```
DB_OK
DB_ERR_FILE_OPEN
DB_ERR_BAD_NAME
DB_ERR_NOT_EXIST
```

Remarks: When a database is deleted, its records are also deleted. In addition, the handles associated with the database, such as the database handle and all the record handles, are made available immediately for reuse.

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

Example:

```
char *name = “timestamps”;
ForgetHandles (name);
DbDelete (name);
```

DbDeleteRec

Deletes a record

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

Parameters: db

This parameter is the handle for the database containing the record. For an orphan record, this parameter is a negative value.

`rec`

The record to be deleted

Return Value: One of the following error codes is returned:

```
DB_OK
DB_ERR_BAD_HANDLE
DB_ERR_NOT_DIR
DB_ERR_IS_DIR
```

Remarks: This function is atomic, that is, the record is either completely deleted from the database or the database is unchanged. No intermediate state exists. This enables data consistency to be preserved if the system stops responding during the function call.

Deleting a record frees the record handle for immediate reuse.

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

DbFileClose

Closes a file

`STATUS DbFileClose( FileType file, BOOL trim )`

Parameters:

`file`

A file number

`BOOL trim`

This parameter is important if the database is extended by `DbFileSeek` or `DbFileWrite`. In this case, the last appended record (of size `newRecSize`) might be only partially written with data and the rest of the record might be filled with bytes consisting of all `1`s.

If `trim` is `TRUE,` then the partially used last record is trimmed. This makes the database length equal to the file `Length` immediately before `DbFileClose`.

If `trim` is `FALSE`, then the partially used last record is not altered, and the database length can be greater than the file length immediately before `DbFileClose`.

Return Value: One of the following error codes is returned:

`DB_OK`
`DB_ERR_FILE_CLOSED`
`DB_ERR_BAD_FILE`

Remarks: The database updates performed by this function or by previous calls to `DbFileSeek` or `DbFileWrite` are not atomic. They are not guaranteed to be permanent until the `DbFileClose` returns `DB_OK`. If the system stops responding during the function call, the database can be left in a partially updated state. The update order is not specified. Some parts might be updated completely, other parts might contain old data, and some parts might contain an arbitrary mix of old and new data. Some bits might be marginally updated, with charge values between 1 and 0. Such bits can return varied results on successive read operations. You must ensure that the data is correctly interpreted.

DbFileInfo

Gets information about a file

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

Parameters: `file`

    A file number

`fileInfo`

    Pointer to a buffer into which the file information should be written

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

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

Return Value: `TRUE` if the file exists, `FALSE` otherwise

DbFileOpen

Opens a file

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

Parameters: `db`

The handle to the database that is to be opened as a streamed file

`newRecSize`

This parameter is the size of new records to be written to the database whenever writing to the file requires extending the underlying database. This value must be between 1 and 65534 (2^{16} - 2), inclusive. At any particular instant, the maximum possible record size can also be limited due to flash memory fragmentation. Refer to `DbMaxNewRecSize`.

Return Value: This function returns a non-negative file number not to be confused with database or record handles. One of the following negative error codes can also be returned:
`DB_ERR_BAD_HANDLE`
`DB_ERR_BAD_SIZE`
`DB_ERR_FILE_OPEN`
`DB_ERR_NOT_DIR`
`DB_ERR_PRIV_HANDLE`
`DB_ERR_NO_FILE`

Remarks: The `DbGetHandle` function must be called prior to calling `DbFileOpen` to obtain the database handle and create the database if it does not exist.

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

DbFileRead

Reads bytes from a file

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

Parameters: `file`

A file number

`data`

Pointer to the destination buffer

`size`

The number of bytes to be read from the file

Return Value: If successful, this function returns the (non-negative) number of bytes actually read. One of the following negative error codes can also be returned:
`DB_ERR_BAD_HANDLE`
`DB_ERR_FILE_CLOSED`

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

The data is read up to the end of the file or until the number of bytes specified by the `size` parameter is read. It is not an error to request to read past the end of the file. If the current position prior to `DbFileRead` is at the end of the file, no bytes are read, and the return value is 0.

DbFileSeek

Sets the current position, possibly extending the file

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

Parameters: `file`

A file number

`pos`

The desired zero-based current position

Return Value: One of the following error codes is returned:

```
DB_OK
DB_ERR_BAD_FILE
DB_ERR_FILE_CLOSED
DB_ERR_NO_HANDLE
DB_ERR_NO_SPACE
```

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

The file extension is subject to the availability of free space required when new records are created. The system does not attempt to create a smaller record if there is not enough space; instead, `DB_ERR_NO_SPACE` is returned.

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

Seeking forward is significantly faster than seeking backward.

DbFileSysInfo

Gets information about the file system

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

Parameters: `FileSysInfo`

Pointer to the buffer to which the file system information should be written

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

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

Example:

```
FileSysInfoType FileSysInfo;
DbFileSysInfo (&FileSysInfo);
NumBefore = FileSysInfo.NumOfClean;

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

DbFileWrite

Writes bytes to a file

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

Parameters:

`file`
: A file number

`data`
: Pointer to the data to be written to the file

`size`
: Size of the data, in bytes

Return Value: One of the following error codes is returned:
`DB_OK`
`DB_ERR_BAD_FILE`
`DB_ERR_FILE_CLOSED`
`DB_ERR_NO_HANDLE`
`DB_ERR_NO_SPACE`

Remarks: The database updates performed by this function are not atomic, and they are not guaranteed to be permanent until the subsequent `DbFileClose` returns `DB_OK`. If the system stops responding, the database might be left in a partially updated state. You must ensure that the data is correctly interpreted.

The file extension is subject to the availability of free space required when new records are created. The system does not attempt to create a smaller record if there is not enough space; instead, `DB_ERR_NO_SPACE` is returned.

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

DbFindNext

Gets a handle to the next database

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

Parameters: db

This parameter is a handle to a database indicating the point after which the database directory is to be searched. A negative value indicates that the search begins with the first directory entry.

`pattern`

This parameter is a pointer to a pattern used to filter the names of the databases in the directory.
Wildcards are indicated by an asterisk (*) and can match any number of characters, including no characters. There are no restrictions on the pattern; it can contain an arbitrary number of wildcard characters alternating with arbitrary text. Within the pattern, an asterisk character is represented as `\*` and the backslash character is represented as `\\`.

Return Value: This function returns a non-negative handle to the next database matching the pattern. One of the following negative error codes can also be returned:
`DB_NO_DB`
`DB_ERR_BAD_HANDLE`
`DB_ERR_NOT_DIR`

Warning! The database handle returned by this function is intended for temporary database identification only, and persists for the lifetime of the database on the particular system. Storing database handles (or record handles) within the permanent data is not advised since the data is not portable; it cannot be meaningfully copied to another system.

Remarks: Directory entries are ordered according to the time of their creation. This order persists after a device resets.

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

DbFirstRec

Gets a handle to the first database record

`HandleType DbFirstRec( HandleType db )`

Parameters: db

A database handle

Return Value: This function returns a non-negative handle to the first record in the database. One of the following negative error codes can also be returned:

```
DB_NO_REC
DB_ERR_BAD_HANDLE
DB_ERR_NOT_DIR
```

Warning! The record handle returned by this function is intended for temporary database identification only and will persist for the lifetime of the database on the particular system. Storing record handles (or database handles) within the permanent data is not advised since the data is not portable; it cannot be meaningfully copied to another system.

Remarks: This function can also be used on the database of directory entries, which returns the handle to the first database entry in the directory.

DbFreeRec

Gets the number of free handles

`unsigned DbFreeRec( void )`

Return Value: The number of unused handles

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

Example:

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

DbFreeSpace

Gets the number of unused bytes

```
unsigned DbFreeSpace( void )
```

Return Value: The total number of unused flash memory bytes in the file system

Remarks: Because of the system overhead and because the flash memory is always fragmented, you cannot create a record as large as the total free space. However, `DbFreeSpace` can be a valuable indicator of space management problems.

Example:

```
if (DbFreeSpace () < MinFreeSpace)
  EmptyUserTrash ();
```

DbGetHandle

Gets a handle to a database, or creates it if one does not exist

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

Parameters: `fileName`

Name of the database to be created or accessed; the maximum length is 65531 (2^{16} - 5).

Return Value: This function returns a non-negative handle to the named database. One of the following negative error codes can also be returned:
`DB_ERR_NO_SPACE`
`DB_ERR_NO_HANDLES`
`DB_ERR_BAD_NAME`

Warning! The database handle returned by this function is intended for temporary database identification only and persists for the lifetime of the database on the particular system. Storing database handles (or record handles) within the permanent data is not advised since the data is not portable; it cannot be meaningfully copied to another system.

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

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

Example:

```
HandleType myDb, newRec;
myDb = DbGetHandle (“timestamps”);

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

DbMaxHandles

Gets the maximum number of record and file handles supported by the system

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

Parameters `MaxRecHandles`

A pointer to a 32-bit integer variable, where the maximum number of record handles is stored; a record handle identifies individual records in the file system, including database header records

`MaxFileHandles`

A pointer to a 32-bit integer variable, where the maximum number of file handles is stored; a file handle identifies a database currently open for streamed access

DbMaxNewRecSize

Gets the maximum size possible for a new record

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

Parameters: NewRec

Where `typedef enum {NORMAL, ORPHAN, DB_NAME} RecKind` is a kind of record, `NORMAL` is a normal database record, `ORPHAN` is an orphan record, and `DB_NAME` is a database directory entry

Return Value: The maximum size possible for a new record of the specified kind, in bytes.

Example:

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

DbName

Gets a database name from a handle

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

Parameters: db

A handle to a database

Return Value: This function returns a pointer to a read-only `NULL`-terminated character string containing the name of the database. If `db` is not a valid database handle, the return value is `NULL`.

Example:

```
// Collect all database names starting with “temp”

HandleType currDb;
currDb = DbFindNext (-1, “temp*”);

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

DbNextRec

Gets a handle to the next database record

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

Parameters: prevRec

A handle to a database record

Return Value: This function returns a non-negative handle to the next record in the database. One of the following negative error codes can also be returned:
`DB_NO_REC`
`DB_ERR_BAD_HANDLE`

Warning! The record handle returned by this function is intended for temporary database identification only and persists for the lifetime of the database on the particular system. Storing record handles (or database handles) within the permanent data is not advised since the data is not portable; it cannot be meaningfully copied to another system.

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

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

DbPointTable

Gets a pointer to the handle mapping table

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

Return Value: Pointer to the handle mapping table

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

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

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

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

DbPointTableEdition

Returns a version number for the `PointTable`.

```
DWORD      __cdecl DbPointTableEdition ();
```

Return Value: Version number for the `PointTable`.

Remarks: Refer to “DbPointTable” on page 150.

DbRecSize

Gets the size of a record

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

Parameters: `rec`

The handle to the record for which the size is requested

`dataOnly`

If `dataOnly` is `TRUE`, the function reports only the size of the user data in the record. If `dataOnly` is `FALSE,` the function also includes the file system overhead.

Return Value: This function returns the non-negative record size, in bytes, or the following negative error code: `DB_ERR_BAD_HANDLE`.

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

DbReplaceOrphan

Replaces a database record with an orphan record

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

Parameters: `rec`

The handle of the record to be replaced

`orphan`

The handle of the orphan record to replace `rec`

Return Value: One of the following error codes is returned:

`DB_OK`
`DB_ERR_NO_SPACE`
`DB_ERR_BAD_HANDLE`
`DB_ERR_IS_DIR`
`DB_ERR_NOT_ORPHAN`

Remarks: This function is atomic, that is, the record is either replaced completely or the database is unchanged. No intermediate state exists. This enables data consistency to be preserved if the system stops responding for any reason during the function call.

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

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

DbReplaceRec

Replaces a database record with a new one

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

Parameters: `rec`

The handle of the record to be replaced

`newSize`

The size of the new data, in bytes. The value must be between 0 and 65534 (2^{16} - 2), inclusive. At any particular instant, the maximum possible record size might be additionally limited due to flash memory fragmentation. Refer to `DbMaxNewRecSize`.

`data`

This parameter is a pointer to the data to be written into the record. If it is `NULL`, the data is assumed to have all bits set to 1. This can be convenient when used in conjunction with the `DbAndRec` function.

Return Value: One of the following error codes is returned:

```
DB_OK
DB_ERR_BAD_SIZE
DB_ERR_NO_SPACE
DB_ERR_BAD_HANDLE
DB_ERR_IS_DIR
```

Remarks: This function is atomic, that is, the record is either completely replaced or completely unchanged. No intermediate state exists. This enables data consistency to be preserved if the system stops responding during the function call.

Alternatively, you can call `DbAndRec` instead of `DbReplaceRec`. `DbAndRec` is faster, but is not atomic.

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

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

DbSecure

Overwrites deleted data with zeroes

```
void DbSecure( void )
```

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

DbSize

Gets database size

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

Parameters: db

A database handle

dataOnly

If `dataOnly` is `TRUE`, the function reports only the size of the user data in the database. If `dataOnly` is `FALSE`, the function also includes the file system overhead in its report.

Return Value: This function returns the non-negative database size, in bytes. One of the following negative error codes might also be returned:

DB_ERR_BAD_HANDLE
DB_ERR_NOT_DIR

Example:

```
// Create indices for sorting databases by size
HandleType handles[MAX_NUM_DB];
int sizes[MAX_NUM_DB];
int i = 0;
HandleType CurrDb = DbFindNext (-1, NULL);
while (CurrDb >= 0) {
  int Temp = DbSize (CurrDb, TRUE);
  assert (Temp >= 0);
  handles[i] = CurrDb;
  sizes[i]   = Temp;
  i++;
  CurrDb = DbFindNext (currDb, NULL);
}
```

ValidHandle

This call checks the validity of the handle parameter.

```
BOOL ValidHandle(HandleType Handle, const void
  *const *PointTable)
```

Parameters: ValidHandle

A database handle

Return Value TRUE if the handle is valid; FALSE if the handle is invalid

File system API error codes

File system functions return the following error codes:

Error Code	Description
DB_OK	The action was completed successfully.
DB_ERR_BAD_HANDLE	The `db`, `rec`, `prevRec` or `orphan` actual parameter is not a valid handle.
DB_ERR_NOT_DIR	The `db` actual parameter is not a valid database directory entry.
DB_ERR_NOT_ORPHAN	The specified record is not an orphan record.
DB_ERR_BOUNDS	The region to be modified does not fit within the existing record.
DB_ERR_IS_DIR	The `rec` parameter refers to a directory entry.
DB_ERR_FILE_OPEN	Streamed file access is currently open on the specified database. It must be closed before deleting the database.
DB_ERR_BAD_NAME	An invalid database name was entered.
DB_ERR_NOT_EXIST	The named database does not exist.
DB_ERR_FILE_CLOSED	The specified file is not open.
DB_ERR_BAD_FILE	The `file` parameter specifies an invalid file number.
DB_ERR_NO_SPACE	There is not enough flash memory to perform the requested action.
DB_NO_REC	The database has no records (not an error).
DB_ERR_BAD_SIZE	The specified size is outside the range of 1 to 65,534 (2^{16}- 2), inclusive.
DB_NO_DB	No databases with matching names exist after the specified database.
DB_ERR_NO_HANDLE	The named database does not exist and there are not enough handles to create it.
DB_ERR_WRONG_DB	The database does not match the record.
DB_ERR_PRIV_HANDLE	The specified handle is privileged.

12
LED API

This chapter provides information on LED API functions.

The LED API provides functions that enable you to control the coverage and message LEDs on the radio modem test board for testing purposes.

LED API functions

RimConfigureLEDs

Enables the LED blinking state to be customized

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

Parameters: `LED_On_Time`

The amount of time, rounded to the nearest 8 milliseconds, during which the blinking LED is on

`LED_Off_Time`

The amount of time, rounded to the nearest 8 milliseconds, during which the blinking LED is off

`DutyCycle`

Percentage of `LED_On_Time` for which the LED is on; the default is 100%

Return Value: No return value

Remarks: The LED blinking states are controlled by this routine so that optimum visibility and contrast can be obtained in the different lighting conditions in which the RIM 902M Radio Modem can be used. Having a low “on” time can also reduce power consumption.

Example: Refer to `DTPING.CPP`.

RimSetLed

Sets the state of the specified LED on the interface test board

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

Parameters: `LedNumber`

You can control two of the LEDs on the interface test board for testing purposes: the wireless network coverage and message LEDs. This parameter specifies which one you are using.

`LedState`

One of: `ON`, `OFF` or `BLINKING`

Return Value: No return value

Remarks: This routine is useful for test purposes while developing, and enables you to reassign up to two of the LEDs on the interface test board for custom uses.

Example: Refer to `DTPING.CPP`.

13
Device events

This chapter explains various types of events that can occur on the device:

- DEVICE_SYSTEM
- DEVICE_TIMER
- DEVICE_RTC
- DEVICE_SERIAL
- DEVICE_RADIO

Note Events that are not documented here may be emitted by the operating system; radio modem applications should ignore these events.

SYSTEM device events

When any of the following events occur, the `Device` member of the `MESSAGE` structure is equal to `DEVICE_SYSTEM`.

SWITCH_BACKGROUND
This event is sent to the current foreground application, indicating that a background application has requested a switch to the foreground. It also indicates that the foreground application has been forced to the background. The `SubMsg` field contains the task handle of the new foreground application.

SWITCH_FOREGROUND
This event is sent to a background application to indicate that it has been moved to the foreground.

POWER_OFF
This event is sent to all applications to indicate that the user is putting the radio modem in a power-off state. Although power is not actually disconnected, the radio is turned off and the applications do not receive any messages. An application can turn off the power by calling `RimRequestPowerOff`.

Note The serial ports, if open at power down, are closed automatically.

POWER_UP
This event is sent to all applications to indicate that the radio modem has left the power-off state. This can be caused by a real-time clock (RTC) alarm event, or a change of state on the serial port control lines, indicating that a serial cable has been plugged in.

TASK_LIST_CHANGED
This message is sent to the foreground task when another task has changed its task switcher information by a call to `RimSetPID`.

MEMORY_LOW
This message is posted when the largest free heap memory block drops below 10 K.

BATTERY_LOW
This message is sent when the power supply voltage drops below 4.0 V.

BATTERY_GOOD
This message is sent when the power supply voltage exceeds 4.1 V.

BATTERY_UPDATE
This message is sent when one of the battery status flags changes.

PERIPHERAL_ID_UPDATE
This message is not implemented on the RIM 902M Radio Modem.

TIMER device events

When any of the following events occur, the `Device` member of the `MESSAGE` structure is equal to `DEVICE_TIMER`.

The RIM 902M Radio Modem contains a pool of 48 global timers for which applications can register using the `RimSetTimer` function. These timers can be configured as either periodic or one-time, with a minimum resolution of 10 milliseconds, and will trigger an event when the timer expires.

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

When a timer expires, a message from `DEVICE_TIMER` is sent to the application. The `Event` field of this message contains the timer ID specified in the call to `RimSetTimer`.

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

To prolong battery life, avoid using short periodic timers (less than five seconds) for long periods.

Real-time clock (RTC) device events

When any of the following events occur, the `Device` member of the `MESSAGE` structure is equal to `DEVICE_RTC`. It is important to note that the RIM 902M Radio Modem has no sense of absolute time, so the RTC events are only meaningful if an application sets the time by some means. The radio modem can measure relative time, which enables it to track how much time has passed since a given event occurred.

RTC_ALARM_EXPIRED
The real-time clock device sends this event to indicate that the alarm time has been reached. This event is sent only to the application that set the alarm.

Note Every application can set a different alarm time. Also see the `RimSetAlarmClock` API call for information on setting the alarm time.

RTC_CLOCK_UPDATE
This event is sent to all applications whenever the date and time are updated. This occurs every minute or when the date and time are changed explicitly. Applications can then call `RimGetDateTime` to receive the current settings.

RADIO device events

When any of the following events occur, the `Device` member of the `MESSAGE` structure is equal to `DEVICE_RADIO`.

MESSAGE_RECEIVED
This event is sent to all applications that have registered to receive radio events (`RadioRegister`). This event indicates that a data packet was received from the Mobitex network.

The `SubMsg` field contains a tag value to be passed into `RadioGetMpak`. `Data[0]` contains the type of the MPAK received, and `Data[1]` contains the HPID (if the MPAK is of type hpdata). Applications should call `RadioGetMpak` to receive the message data.

MESSAGE_SENT
This event is an acknowledgement that a transmitted packet was received by the Mobitex network. This event is sent to the application that sent the packet, whether that application is in the foreground or the background. The `SubMsg` field contains the tag value that was returned by `RadioGetMpak`.

MESSAGE_NOT_SENT
This event indicates that an attempt to transmit information to the Mobitex network failed. This event is sent to the task that submitted the packet when coverage is too poor for transmission or when an invalid data package is sent. The `SubMsg` field contains the tag value returned by `RadioGetMpak`. The `Data[0]` contains the error number.

SIGNAL_LEVEL
This event is sent to all registered applications to indicate that the received signal level has changed. The SubMsg field contains a negative value that represents the level of the signal in dBm. A more positive value (closer to zero) indicates a stronger signal. A value of -256 dBm (`RSSI_NO_COVERAGE`) indicates that the modem is out of coverage.

NETWORK_STARTED
This event is sent to all registered applications to indicate that the radio modem has been turned on or has just switched to a new network.

BASE_STATION_CHANGE
This event is sent to all registered applications to indicate that the radio modem has been turned on or has just switched to a new network.

RADIO_TURNED_OFF
This event is sent to all registered applications to indicate that the radio modem has been turned off, either by the user or as a result of a low battery.

MESSAGE_STATUS
Due to the queuing of packets in the radio API layer, and the retry spacing of the radio code, an MPAK sent to the Radio API might not be transmitted immediately. The sender of an MPAK is notified of that MPAK transmission status through this event. The `Data[0]` field of the message structure contains one of the following status subcodes:

- `MPAK_TRANSMITTING`
  The MPAK is currently being sent by the radio.
- `MPAK_TX_PENDING`
  The radio code is currently not transmitting the MPAK because of transmission difficulties; it will try again later.

Serial device events

When any of the following events occur, the `Device` member of the `MESSAGE` structure is equal to `DEVICE_SERIAL`.

Note Events are always sent to the task that opened the communications port. Applications should ignore any events emitted by the radio modem that are not documented here.

SERIAL_DTR_CHANGE
This event indicates a level change on the DTR serial port input. The new state of the control line is placed in the `SubMsg` field.

SERIAL_RX_ERROR
If a communications error, such as receive overrun, framing error, or a parity error is received, a `SERIAL_RX_ERROR` message is sent to the task that opened the specified serial port. The `SubMsg` field of the message contains the specific error number.

SERIAL_RX_AVAILABLE
This event indicates that the serial receive queue has changed from empty to not empty. Only one `SERIAL_RX_AVAILABLE` message is sent each time that the serial buffer changes from empty to not empty. Thus, an application does not receive another `SERIAL_RX_AVAILABLE` message until it has completely emptied the serial buffer and new characters are received. Refer to `SerialRead` for more details.

SERIAL_TX_EMPTY
A `SERIAL_TX_EMPTY` event is sent to the task that opened the specified serial port whenever the port's transmit FIFO, as well as the serial transmit hardware, becomes completely empty.

SERIAL_RTS_CHANGE
This event indicates a level change on the RTS serial port input. The new state of the control line is placed in the SubMsg field.

POWER_OFF
When the radio modem is turned off while an application has a serial port open, the serial port is closed automatically. If the application wants to resume use of the serial port when the radio modem is turned on again, it must open the port again. The `POWER_OFF` event is not specific to a serial port, but applications that use serial ports must be aware of the effects of being turned off.

Appendix

Compatibility of C library functions

Certain functions within the compiler C library are safe to call from the radio modem environment. Other functions, however, are not compatible with the radio modem. The following list details the C functions that can and cannot be used.

Functions that are compatible

Argument access macros

This set of functions is compatible with the radio modem environment. This includes the macros `va_arg`, `va_start`, and `va_end`.

Buffer manipulation functions

This set of functions is compatible with the radio modem environment. This includes `memccpy`, `memchr`, `memcmp`, `memcpy`, `_memicmp`, `memmove`, `memset`, and `_swap`.

Data conversion functions

The following data conversion functions can be used in a radio modem application: `abs`, `atoi`, `_atoi64`, `atol`, `_itoa`, `_i64toa`, `labs`, `_ltoa`, `strtol`, `strtoul`, `__toascii`, `tolower`, `_tolower`, `toupper`, `_toupper`, and `_ultoa`.

Searching and sorting functions

The library functions `bsearch`, `_lfind`, `_lsearch`, and `qsort` should work in the radio modem environment.

Functions that are not compatible

Byte classification (multibyte) functions

The radio modem does not support multibyte characters.

Character classification functions

Since the radio modem 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 radio modem does not support the compiler library debugging functions. Other functions are provided to support debugging radio modem 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 radio modem file system is different from those used on desktop systems. As a result, the directory control functions are not supported by the radio modem.

Error and exception handling functions

The radio modem environment does not support exception handling. Serious failures, such as page faults, are handled by the system function `RimCatastrophicFailure`, which might also be called by application code when an unrecoverable error is detected.

File handling functions

The radio modem file system is different from those used on desktop systems. As a result, the file handling functions are not supported by the radio modem.

Floating point functions

Since the radio modem has no floating point coprocessor and does not support application handling of traps and exceptions, floating point functions cannot be used on the radio modem, unless all of the floating points are implemented by emulation without calls to the operating system. However, the following functions are incorrectly classified as floating point functions, and may be used without difficulty: `div`, `labs`, `ldiv`, `_lrotl`, `_lrotr`, `_max`, `_min`, `rand`, `_rotl`, and `_rotr`.

Input and output functions

Any stream, file, or console I/O functions are not compatible with the radio modem environment. Because of their compiler library implementation, this category also includes `sprintf` and `sscanf`. These cannot be used in a radio modem 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 radio modem applications run in a protected environment, calling these functions cause a protection fault at run time.

Locale-dependent functions

The radio modem environment does not support locales, multibyte characters, or wide characters. However, many of the locale-dependent functions are compatible. These functions are those that were in the C library before the `setlocale` function.

Memory allocation functions

The standard C memory allocation functions cannot be used in the radio modem 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. Operator new differs from the standard one in that, if there is insufficient memory to perform the allocation, it returns a null pointer rather than throwing an exception.

Process and environment control functions

Because the process model of the radio modem 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. It is possible to write a version of these functions that work on the radio modem, if it is acceptable not to call the distracters 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 radio modem environment.

System call functions
Windows system call functions cannot be used in the radio modem environment. Because of the way in which the radio modem operating system allocates application stacks, Windows functions are not safe to call even when running them on the simulator.

Time functions
The radio modem operating system represents time differently than do desktop systems. As a result, time functions are not compatible with the radio modem environment.

I²C Driver

I²C is a software add-on, supported by the RIM 902M Radio Modem. It requires the use of any two of the four extra I/O lines, pins 1 to 4. This sample implementation makes no attempt at a consistent clock speed, and leave the bus busy if an interrupt causes other processing to be performed. This sample implementation does not support multimaster arbitration and synchronization, but could be modified to do so. The clock is a true open-collector clock.

For more information about I²C, visit:

`http://www.calibreuk.com/i2c2000/i2capp.htm`

The I²C driver files that are provided are meant to be a sample of how you might implement I²C using the RIM radio modems. They are meant to be configured or modified as required.

Definitions for the sample I²C driver

```
#ifndef __I2C_H
#define __I2C_H

#include "basetype.h"
```

Port definitions for user bits

```
#define PINDIR0     0xF861
#define PINLTC0     0xF862
#define PINSTA0     0xF863
```

User bit definitions:

```
#define USER01      0x08
#define USER02      0x04
#define USER03      0x02
#define USER04      0x01
```

Change the following to match your hardware design:

`#define SCL`	`USER01`	I²C clock line
`#define SDA`	`USER02`	I²C data line

Function to initialize the I²C software and hardware:

```
void I2C_init(void);
```

The following function performs a write operation followed by a read operation on the I²C bus. This function returns TRUE if successful, and returns FALSE if an error occurs.

```
BOOL I2C( BYTE SlaveAddress, //Even number between 8 and 238
  const BYTE *OutData, // Can be NULL if NumOutBytes is zero
  int        NumOutBytes,// Number of bytes to send
  BYTE      *InData, //Can be NULL if NumInBytes is zero
  int        NumInBytes ;) //Number of bytes to read
#endif //__I2C_H
```

Sample I²C driver

```
#include “I2C.h”
#pragma intrinsic(inp, outp)
#define RW_BIT  0x01

static BYTE OtherBits; // keeps state of non-I²C bits in same byte
static BOOL I2C_Error; // flag set if error occurs

void I2C_init(void)
{
  // Step 1. Configure the SCL and SDA bits as open drain outputs.
  // Initially set high.
  outp( PINDIR0, inp( PINDIR0 ) | SDA | SCL );
  outp( PINLTC0, inp( PINLTC0 ) | SDA | SCL );
}
static void Delay(void)
{
    __asm   nop
}
```

```
static void I2C_SendByte(int data)
{
  int bitmask;

  for (bitmask = 0x80; bitmask != 0; bitmask >>= 1) {
    int databit = data & bitmask ? ~0 : ~SDA;
    outp( PINLTC0, OtherBits & databit & ~SCL );
    Delay();// output data with low clock
    outp( PINLTC0, OtherBits & databit );

    //drive clock high
    while( (inp(PINSTA0) & SCL) == 0 );
    Delay();//wait for clock to go high
    outp( PINLTC0, OtherBits & databit & ~SCL );
  }

  Delay();//drop clock
  outp( PINLTC0, OtherBits & ~SCL );
  Delay();//stop driving data bit

  //drive clock high for acknowledgemet pulse
  outp( PINLTC0, OtherBits );
  while( (inp(PINSTA0) & SCL) == 0 );
  Delay();//wait for clock to go high
  if ((inp(PINSTA0) & SDA) != 0) {
    I2C_Error = TRUE;
  }// read acknowledge pulse status

  outp( PINLTC0, OtherBits & ~SCL ); }
}//drop clock

static BYTE I2C_ReadByte(BOOL LastByte)
{
  int bitmask;
  int data = 0;

  for (bitmask = 0x80; bitmask != 0; bitmask >>= 1) {
    outp( PINLTC0, OtherBits & ~SCL );
    Delay();//drive clock low, float data line
    outp( PINLTC0, OtherBits ); //drive clock high
    while( (inp(PINSTA0) & SCL) == 0 );
    Delay();
    if (inp(PINSTA0) & SDA) {
      data |= bitmask;
    }//wait for clock to go high
    outp( PINLTC0, OtherBits & ~SCL );
  }
  bitmask = LastByte ? ~0 : ~SDA;//drop clock
  outp( PINLTC0, OtherBits & bitmask & ~SCL );
  Delay();// drive data low or high
  //drive clock high for acknowledge pulse
  outp( PINLTC0, OtherBits & bitmask );


  while( (inp(PINSTA0) & SCL) == 0 );
  Delay();//wait for clock to go high
  outp( PINLTC0, OtherBits & bitmask & ~SCL );
  return (BYTE) data;
}// drop clock

BOOL I2C(BYTE SlaveAddress, const BYTE *OutData, int NumOutBytes,
  BYTE *InData, int NumInBytes)
{
  I2C_Error = FALSE;
  OtherBits = (BYTE) inp(PINLTC0);

  if (NumOutBytes == 0 && NumInBytes == 0) {
    // Erroneous request!
    return FALSE;
  }
  if ((OtherBits & (SCL | SDA)) != (SCL | SDA)) {
    // I2C lines are being controlled by some other software!
    return FALSE;
  }
  if ((inp(PINSTA0) & (SCL | SDA)) != (SCL | SDA)) {
    // I2C lines are being pulled low!
    return FALSE;
  }

  // Step 2. Generate a start condition
  outp( PINLTC0, OtherBits & ~SDA );
  Delay();
  outp( PINLTC0, OtherBits & ~SDA & ~SCL );

  //Step 3. Perform output transfer
  if (NumOutBytes != 0) {
    int i;
    I2C_SendByte(SlaveAddress & ~RW_BIT);
    for (i = 0; i < NumOutBytes && !I2C_Error; i++) {
      I2C_SendByte(*OutData++);
    }
  }

  //Generate repeated start condition, if necessary
  if (NumOutBytes != 0 && NumInBytes != 0 && !I2C_Error) {
    Delay();
    outp( PINLTC0, OtherBits & ~SDA );
    Delay();
    outp( PINLTC0, OtherBits & ~SDA & ~SCL );
  }
  //Step 4. Perform input transfer
  if (NumInBytes != 0 && !I2C_Error) {
    int i;
    I2C_SendByte(SlaveAddress | RW_BIT);
    for (i=1; i<NumInBytes; i++) {
      *InData++ = I2C_ReadByte(FALSE);
    }
    *InData++ = I2C_ReadByte(TRUE);
  }

  //Step 5. Generate stop condition
  outp( PINLTC0, OtherBits & ~SDA & ~SCL );
  Delay();
  outp( PINLTC0, OtherBits & ~SDA );
  Delay();
  outp( PINLTC0, OtherBits );
  return (I2C_Error == FALSE);
}
```

An I²C_demo

This is a skeleton demonstration to illustrate calling the I²C driver. You can adapt this sample as required.

A sample session follows:

9CS: I²C address set	“9C” enters a hex number “S” sets it as the address
1FW: Data written	“1F” enters a data value, “W” writes it to the device
R: 1F	“R” reads a data byte and outputs it to the serial port

```
#include "RimOEM.h"
#include "I2C.h"

#define PORT    0

//step 1. Define variables for all applications.
char VersionPtr[] = "I2C Demo";
int AppStackSize = 512;

//Step 2. The application entry point is called PagerMain
void PagerMain(void)
```

```
{
  static const struct SerialOpenParms OpenParms = {
    9600, 8, SERIAL_NO_PARITY, 1, 2048, 8192,
        SERIAL_NO_FLOW_CONTROL
  };
  MESSAGE msg;
  BYTE buf = 0, ch = 0, i2c_addr = 0, prev_ch;

  // Step 3. Initialize the serial port.  This must be done before
  // the first RimGetMessage to signal that it should
  // not take over the serial port.

  // In a real program, you should check the return code in
  // case the serial port cannot be opened.

  SerialOpen(PORT, &OpenParms, sizeof(OpenParms));

  //Step 4. Initialize the I2C subsystem
  I2C_init();

  //Step 5. Message loop
  for(;;) {
    RimGetMessage( &msg );
    switch (msg.Event) {
      case POWER_UP:

        // If the device has been turned off and turned on
        // again, you must re-open the serial port. If this has not
        // occurred, proceed to the next step.

        SerialOpen(PORT, &OpenParms, sizeof(OpenParms));
        break;
      case SERIAL_RX_AVAILABLE:
        prev_ch = ch;
        ch = (BYTE) SerialReadChar(PORT);
        switch(ch) {
          case '0':
          case '1':
          case '2':
          case '3':
          case '4':
          case '5':
          case '6':
          case '7':
          case '8':
          case '9':
            SerialSendChar(PORT, ch);
            buf = (BYTE) ((buf << 4) | (ch - '0'));
            break;
```

```
case 'a':
case 'b':
case 'c':
case 'd':
case 'e':
case 'f':
case 'A':
case 'B':
case 'C':
case 'D':
case 'E':
case 'F':
  SerialSendChar(PORT, ch);
  buf = (BYTE) ((buf << 4) | ((ch & 0x0F) + 9));
  break;

case 's':
case 'S':
  //Step 6. Set I2C device address
  i2c_addr = buf;
  SerialSend(PORT, (BYTE *) "S:I2C address set\r\n", 20);
  buf = 0;
  break;

case 'w':
case 'W':
  // Step 7. Write the specified data byte via I2C to the
  // device specified in the previous step. Check for and
  // print out some basic diagnostics.
  SerialSendChar(PORT, ch);

  if ((inp(PINLTC0) & (SCL | SDA)) != (SCL | SDA)) {
    SerialSend(PORT, (BYTE *)
    "\r\nLines controlled by other software!\r\n", 43);
    break;
  }
  if ((inp(PINSTA0) & (SCL | SDA)) != (SCL | SDA)) {
    SerialSend(PORT, (BYTE *) "\r\n", 2);
    if ((inp(PINSTA0) & SCL) != SCL) {
      SerialSend(PORT, (BYTE *)
        "SCL line is being pulled low!\r\n", 31);
    }
    if ((inp(PINSTA0) & SDA) != SDA) {
      SerialSend(PORT, (BYTE *)
        "SDA line is being pulled low!\r\n", 31);
    }
    break;
  }

  if (I2C(i2c_addr, &buf, 1, NULL, 0)) {
    SerialSend(PORT, (BYTE *) ": Data written\r\n", 16);
  } else {
    SerialSend(PORT, (BYTE *) ": I2C Error!\r\n", 14);
  }
  buf = 0;
  break;

case 'r':
case 'R':
  //Step 8. Read a byte via I2C and display it.
  if (I2C(i2c_addr, NULL, 0, &buf, 1)) {
    SerialSend(PORT, (BYTE *) "R: ", 3);
    SerialSendChar(PORT, (BYTE)
      "0123456789ABCDEF"[(buf >> 4) & 0x0F]);
    SerialSendChar(PORT, (BYTE)
      "0123456789ABCDEF"[(buf >> 0) & 0x0F]);
    SerialSend(PORT, (BYTE *) "\r\n", 2);
  } else {
    SerialSend(PORT, (BYTE *) "R: I2C Error!\r\n", 15);
  }
  buf = 0;
  break;
case '?':

  // Debugging aid. Print buffer value
  SerialSend(PORT, (BYTE *) "?: ", 3);
  SerialSendChar(PORT, (BYTE)
    "0123456789ABCDEF"[(buf >> 4) & 0x0F]);
  SerialSendChar(PORT, (BYTE)
    "0123456789ABCDEF"[(buf >> 0) & 0x0F]);
  SerialSend(PORT, (BYTE *) "\r\n", 2);
  break;

case 0x0E:
case 0xAA:
case 0x0F:
case 0xF0:

// Do not echo these characters. They may be part of
// a reset request.

// Step 9. Do a simple check, then reset to load new
// code
if (ch == 0xF0 && prev_ch == '\x0F') {
  RimInitiateReset();
}
break;
```

```
        default:
          //Step 10. If there is an unknown code, echo with ?
          SerialSendChar(PORT, ch);
          SerialSend(PORT, (BYTE *) "?\r\n", 3);
          break;
      }
    break;
    }
  }
}
```

Index