BlackBerry Software Development Kit

Version 2.5

Radio API Reference Guide (DataTAC)

BlackBerry Software Development Kit Version 2.5 Radio API Reference Guide
Last modified: 18 March 2002
Part number: PDF-04639-001

At the time of printing, this documentation complies with RIM Wireless Handheld Version 2.5.

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

Produced in Canada

Contents

About this guide

This guide provides a detailed reference for the Radio Application Programming Interface (API).

The Radio API provides packet-level access to the DataTAC network using function calls to send and receive data. You do not need extensive knowledge of the network to use these functions.

This guide assumes you have experience with C++ programming.

Other resources

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

All RIM documentation is available at `http://developers.rim.net`.

- *BlackBerry SDK Developer Guide*

  This guide explains how to use the BlackBerry SDK and contains sample code for the wireless handheld's general functions.

- *BlackBerry SDK Message API Reference Guide*

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

- `README.txt`

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

Chapter 1 Getting started

This chapter provides an overview of radio communications over the DataTAC network, including these topics:

- overview of the DataTAC network
- how data is routed in the network

Note: This chapter provides background information on radio communications on the DataTAC network. This information is not intended to be comprehensive. Contact your network operator for complete documentation on network operation.

Understanding the DataTAC network

DataTAC is a packet-switched, wide-area wireless data network. DataTAC is one of the world’s most widely available narrowband personal communications service (PCS) networks. The network infrastructure uses digital cellular service providers around the world. Motient operates the DataTAC network in the United States and Bell Mobility operates the DataTAC network in Canada. DataTAC service is also available throughout Asia, Australia, and Europe.

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.

DataTAC provides highly reliable, 2-way digital data transmission. The network provides error detection and correction, including transmission acknowledgement, to maintain the integrity of the data being sent.

Packet-switching technology provides 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

Each RIM Wireless Handheld, and any other device on the DataTAC network, is assigned a unique Logical Link Identifier (LLI).

DataTAC offers two basic routing methods:

- host-based routing
- peer-to-peer routing

Host-based routing

Host-based routing assumes that a central computer processes applications that service many wireless devices, such as the RIM Wireless Handheld.

Host-based routing is used for communication between a mobile device and one or more servers in a fixed location. The server is normally connected to the DataTAC network over a land line, rather than radio frequencies. This provides a more economical means of communication when many packets are transmitted.

The following are some of the environments that are suited to host-based routing:

- when a mobile device has access to information stored in a database located on a remote server
- when a server performs much of the processing
- when the mobile unit fills in fields of a form that is located on the server and then transmits those fields

The handheld radio modem uses the Radio Data Link Access Protocol (RD-LAP) to transfer PDUs between the handheld and a DataTAC base station. RD-LAP transfers data at either 9600 bits per second (bps) or 19,200 bps, depending on the network. Some DataTAC networks also offer the Mobile Data Communications (MDC) protocol, which operates at 4800 bps. Your DataTAC network operator can provide you with details of the radio link protocol(s) in use.

Peer-to-peer routing

Peer-to-peer routing is used for communication between two mobile devices. A Message Generator (MG) host on the DataTAC network manages all the details of the communication.

The Message Generator (MG) record format is used for peer-to-peer routing. MG is the application user header of the service data unit (SDU). The MG header signals to the DataTAC network that the SDU originates from a handheld and is destined to another device. When the SDU is delivered, the MG in the user header is replaced with Received Message (RM).

Refer to the following file included with the SDK: `...\OS_Samples\src\dt_ping.c`.

Data packets

The data stream between devices connected to the DataTAC network is divided into SDUs. SDUs can contain up to 2048 bytes for routing information (such as a destination address) and user data.

The application must build and format SDUs. When sending more than 2048 bytes of data, an application must divide data into several SDUs of up to 2048 bytes each. For example, a 5000-byte “message” must be sent to the handheld as three data packets: two packets of 2048 bytes, and a final packet with 904 bytes of user data. The application at the destination must collect these packets, verify that none are missing, and place them in the correct sequence. This might require a transport layer.

The `SDU_HEADER` is defined in `datatac.h`. Not all of the information about the SDU is encapsulated in the `SDU_HEADER` structure. For example, the LLI for the SDU destination is not encoded in this structure. The structure of the data packet varies greatly between network providers. Contact your network operator for information on addressing and routing SDUs.

Note: At the data link layer between the handheld and the network base station, each SDU is divided into smaller 512-byte or 256-byte protocol data units (PDUs). These packets are sent one at a time over the radio network and are then reassembled at the destination into the SDU. If your network operator charges airtime based on the number of PDUs transmitted, it might be economical to design applications to minimize the number of PDUs, instead of the number of bytes. Contact your DataTAC network operator to discuss airtime pricing and how to develop an application that minimizes data transmission cost.

SDU formatting

When your application constructs an SDU, it must fill in the `SDU_HEADER` structure and provide a pointer to the data. Refer to "Structures" on page 11 for more information. Refer to the samples `dt_ping.c` and `hostsim.c` included with the SDK for examples. The following table describes the format of an MG message for peer-to-peer routing over the DataTAC network. Optional fields are shaded.

Byte	Description
2	User header length, most significant byte first (must set to 0x0002)
2	User header (MG if sending, RM if received)
8	LLI of sender (Hex-ASCII)
1	Forward slash delimiter (0x2F)
8	LLI of destination #1 (Hex-ASCII)
1	Forward slash delimiter (0x2F) - *optional*
8	LLI of destination #2 (Hex-ASCII) – *optional*
1	Forward slash delimiter (0x2F) – *optional*
8	LLI of destination #3 (Hex-ASCII) – *optional*
1	Forward slash delimiter (0x2F) – *optional*
8	LLI of destination #4 (Hex-ASCII) - *optional*
1	Carriage return delimiter (0x0D)
1	Bit 6 = TRUE: reply to a requested reply
1	Bit 7 = TRUE: reply message Bit 6 = TRUE: reply requested Bit 5 = TRUE: forwarded message Bit 3 = TRUE: registered receipt Bit 2 = TRUE: registered message Bit 1 = TRUE: priority message
5	Month, day, year, hour, minute (binary-coded decimal)
8	Original source LLI (Hex-ASCII) - *only if forwarded message*
1	Carriage return delimiter (0x0D)
~	Data area (variable length)

Chapter 2
Radio API reference

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

DataTAC Radio API

The Radio API provides access to the radio network using simple API function calls to send and receive data. You do not need extensive knowledge of the radio 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. Refer to "Radio events" on page 23 for more information.

Structures

The Radio API uses the following structures.

SDU_HEADER

This structure represents the header data in an SDU packet.

```
typedef struct {
    WORD  HeaderLength;
    BYTE  Resend;
    BYTE  Priority;
} SDU_HEADER;
```

Field	Description
`headerlength`	Indicates the length of the header data, also called the user header offset (UHO).
`resend`	Indicates whether the SDU is a retry of the previous SDU: SEND_NEW SEND_RESENDS
`priority`	Indicates the priority level of the SDU: PRI_NORMAL PRI_VERY_LOW PRI_LOW PRI_HIGH

RADIO_INFO

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

```
typedef struct {
    SDWORD RadioOn;
    DWORD  LLI;
    DWORD  ESN;
    DWORD  FaultBits;
    SDWORD RSSI;
    BYTE   Area;
    BYTE   Base;
    WORD   ChannelDesignator;
    BYTE   ChannelAttributes[2];
    WORD   RfVersion;
    WORD   RcVersion;
    BYTE   Reserved[2];
    BOOL   Contact;
    DWORD  PowerSaveMode;
    BOOL   ReceiverEnabled;
    BOOL   TransmitterEnabled;
} RADIO_INFO;
```

Field	Description
`RadioOn`	Either RADIO_ON or RADIO_OFF.
`LLI`	The LLI as a 32-bit value.
`ESN`	The electronic serial number, as a 32-bit value.
`FaultBits`	Indicates a series of flags indicating possible problems on the handheld.

Field	Description
`RSSI`	A value ranging from –113 to –40, or RSSI_NO_COVERAGE; values above –90 are generally reliable coverage.
`Area/Base`	Current Base and Area IDs for that location in which the handheld has network coverage, or for the location in which the handheld last had network coverage.
`ChannelDesignator`	Indicates which channel the handheld is currently using.
`ChannelAttributes`	Indicates the attributes of the channel the handheld is currently using.
`RfVersion`	Version of the protocol currently in use.
`RcVersion`	Version number of device radio code.
`Contact`	Indicates whether the handheld is in contact with the network (1 = in coverage, 0 = out of coverage).
`PowerSaveMode`	Indicates the status of the powersave mode: 0 = EXPRESS 1 = MAXIMUM 2 = AVERAGE 3 = MINIMUM
`ReceiverEnabled`	Indicates the status of the receiver mode: (1 = Rx enabled, 0 = Rx disabled).
`TransmitterEnabled`	Indicates the status of the transmitter mode (1 = Tx enabled, 0 = Tx disabled).

NETWORKS_INFO

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

```
typedef struct {
    int   DefaultNetworkIndex;
    int   CurrentNetworkIndex;
    int   NumValidNetworks;
    struct {
                 BYTE    NetworkName[10];
                 BYTE    NetworkAddress[3];
                 BYTE    NetworkType;
                 BYTE    NetworkChannels;
                 BYTE    NetworkLLIs;
    } 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 US) Part of the `Networks` substructure in NETWORKS_INFO
`NetworkName[10]`	Name of the network Part of the `Networks` substructure in NETWORKS_INFO
`NetworkAddress[3]`	Network home address
`NetworkType`	Network type: 0 - DataTAC 4000 1 - DataTAC 5000 2 - DataTAC 6000
`NetworkChannels`	Number of channels for network
`NetworkLLIs`	Number of group LLIs for network

Functions

The following functions are listed alphabetically.

RadioAccelerateRetries

Causes the radio to retry transmitting more aggressively.

```
void RadioAccelerateRetries(int sduTag)
```

Parameters	`sduTag`	Tag of the MPAK for which the radio should accelerate transmission (not currently used).

Description 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 handheld 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).

RadioCancelSendSdu

Cancels a submitted SDU.

```
int RadioCancelSendSdu(int sduTag)
```

Parameters	`sduTag`	This is the tag assigned by the application server when the packet is submitted to the handheld for transmission. Passing in `-1` for the `sduTag` value cancels all SDUs queued for transmission by the calling application.

Returns The number of SDUs that are cancelled; a negative value if an error occurs.

Description The `RadioCancelSendSdu` attempts to cancel a submitted packet that is identified by the tag number.

If `RadioCancelSendSdu` is called before the SDU is transmitted, the SDU is returned to the application as cancelled, provided that it has not already been sent. There is no guarantee, however, that a cancelled SDU was not already received by the DataTAC network.

RadioChangeNetworks

Changes the current radio network.

```
bool RadioChangeNetworks(BYTE * NetworkName)
```

Parameters

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

Returns No return value.

Description 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.

RadioChangePowerSave

Changes the enhanced powersave (EPS) mode.

```
int RadioChangePowerSave(int mode)
```

Parameters

`mode`	The EPS mode to use, where `mode` is one of the integers listed in the following table.

Int	Mode	Description
0	Express	Express mode leaves the receiver on continuously. This means that power consumption is extremely high, but latency in receiving packets is eliminated. This mode should be used only by applications that are not powered by batteries and for which power consumption is not a consideration.
1	Maximum	Maximum mode (EPS 1) provides the maximum battery life possible. The receiver turns on automatically once every two minutes to check for buffered messages at the base station. If any messages are pending to the radio, the receiver stays on to receive them.
2	Average	Average mode (EPS 2) consumes more power than EPS 1, but reduces the reception latency to one minute.
3	Minimum	Minimum mode (EPS 3) consumes more power than EPS 2 but reduces reception latency to 30 seconds.

Returns If the powersave mode is changed successfully, this function returns 0; otherwise, the function returns a negative value (refer to "Radio events" on page 23).

Description Using powersave modes only affects the latency for receiving data. Powersave does not increase the latency of transmitting data from the radio.

RadioDeregister

Deregisters applications from receiving radio events.

```
void RadioDeregister(void)
```

Returns No return value.

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

Example Refer to `DTPING.CPP`.

RadioGetAvailableNetworks

Programs the available networks into the handheld.

```
void RadioGetAvailableNetworks(NETWORKS_INFO * info)
```

Parameters

`info`	Pointer to a NETWORKS_INFO structure (refer to "Structures" on page 11).

Returns No return value.

Description Enables you to query the handheld and determine which networks have been programmed.

RadioGetDetailedInfo

Retrieves the current state of the radio.

```
void RadioGetDetailedInfo(RADIO_INFO * info)
```

Returns No return value.

Description Retrieves the current state of the radio, such as LLI, RSSI, on/off, powersave/express, base, and area, into a `RADIO_INFO` structure (refer to “Structures” on page 11 for details)

Example Refer to `DTPING.CPP`.

RadioGetSdu

Retrieves the data when an SDU is received.

```
int RadioGetSdu(int sduTag,
    SDU_HEADER * header,
    BYTE * data)
```

Parameters

`sduTag`	This argument is the `SDU_TAG` value from the `MESSAGE_RECEIVED` message. Note that the `SDU_TAG` value has a limited life span. For received SDUs, the tag must be used before getting the next message or yielding control.
`header`	This argument is a pointer to an `SDU_HEADER` structure (refer to “Structures” on page 11 for details). The header information extracted from the SDU will be placed in this structure. The `header length` element of the `HEADER` structure contains the user header length of the SDU. The `Resend` element of the `HEADER` structure contains the SDU resend information. The `Priority` element of the `HEADER` structure contains the SDU priority information.
`data`	This argument is a pointer to a buffer that is large enough to contain the SDU data (at least 2048 bytes). The amount of space required can be determined in advance by calling `RadioGetSdu`. If the `data` pointer is `NULL`, the SDU data is not copied.

Returns If the function is successful, the return value is the number of data bytes in the data portion of the SDU (0 to 2048). If the function is unsuccessful, the return value is negative.

Description When an SDU is received, the `sduTag` value is contained in the message. This tag value is used to obtain subsequent information about the SDU.

`RadioGetSdu` can also be used to get copies of SDUs that are queued for transmission. SDUs that are queued for transmission can be recalled at any time until the first `RimTaskYield` or `RimGetMessage` after the SDU is returned to the application as either sent or unsuccessful.

`RadioGetSdu` can be used in several ways:

- Get header only: to obtain only the header set, the data pointer to NULL
- Get Header and SDU: both pointers point to their respective data areas; only the data portion of the SDU is copied into the data buffer.

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

Example Refer to `DTPING.CPP`.

RadioGetSignalLevel

Gets the current signal strength.

```
int RadioGetSignalLevel()
```

Returns Radio signal level in dBm, if the handheld is in an area of wireless network coverage; the value is typically between -121 dBm and -40 dBm.

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

Description 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 Refer to `DTPING.CPP`.

```
// Displays the strength of the 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 state

Returns The function returns the state of the radio before `RadioOnOff` was called, and can be one of the following 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.

Description 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.

Example Refer to `DTPING.CPP`.

RadioRegister

Registers applications for radio events.

```
void RadioRegister()
```

Returns No return value.

Description 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.

Example Refer to `DTPING.CPP`.

RadioResumeReception

Indicates that the application is ready to receive MPAKs again.

```
void RadioResumeReception()
```

Returns No return value.

Description 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 MPAK is 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.

Example Refer to `DTPING.CPP`.

RadioSendSdu

Submits an SDU for transmission by the radio.

```
int RadioSendSdu(SDU_HEADER * header,
    BYTE * data,
    int length)
```

Parameters

`header`	A pointer to an SDU_HEADER structure that contains information for building the SDU header.
`data`	A buffer containing the data bytes to be included in the SDU.
`length`	Indicates the number of data bytes to be included in the SDU.

Returns A tag is assigned to the SDU by the application server. If the sequence identification is negative, the message could not be queued for sending. The returned tag value is always less than `MAX_QUEUED_SDUS`, which is currently 7.

Description `RadioSendSdu` submits an SDU for transmission by the radio. If an SDU has already been submitted for transmission by this or any other application, the SDU is queued for transmission. If more than four SDUs are already queued, `RadioSendSdu` fails and returns a negative error code.

`RadioSendSdu` copies the data provided, so the data pointed to when the call is made can be deleted after the call returns.

Example Refer to `DTPING.CPP`.

RadioStopReception

Indicates that the radio is not ready to receive MPAKs.

`void RadioStopReception(int sduTag)`

The `RadioStopReception` stops the receiving of SDUs for the pager. It is intended to be used if all buffers for receiving SDUs are full.

Parameters

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

Description The `RadioStopReception` function stops the handheld 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 handheld.

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.

Example Refer to `DTPING.CPP`

Radio events

When any of the following events occur, the `Device` member of the `MESSAGE` structure is equal to `DEVICE_RADIO`.

Event	Description
`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 DataTAC network. The `SubMsg` field contains a tag value to be passed into `RadioGetSdu`. `RadioGetSdu`
`MESSAGE_SENT`	This event is an acknowledgement that a transmitted packet was received by the DataTAC 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 `RadioGetSdu`.
`MESSAGE_NOT_SENT`	This event indicates that an attempt to transmit information to the DataTAC 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 `RadioGetSdu`. 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.

Event	Description
BASE_STATION_CHANGE	This event is sent when the handheld switches base stations. It has no other effect and requires no action on the part of the application.
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	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 : MPAK is being sent by the radio • MPAK_TX_PENDING: The radio is not transmitting the MPAK because of transmission difficulties; it will try again later

Error codes

The following error codes pertain to radio function return values.

Code	Description	Description
-1	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.
-2	RADIO_SDU_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.
-3	RADIO_NO_FREE_BUFFERS	Attempting to send an MPAK with all the radio's outgoing buffers full produces this error code.
-4	RADIO_BAD_DATA	Attempting to send an MPAK with format data that cannot be used in an MPAK produces this error code.
-5	RADIO_BAD_TAG	Attempting to fetch an MPAK with a tag value outside the legal range produces this error code.
-6	RADIO_ERROR_GENERAL	This is a generic radio error.

Index