IridiumSBD Arduino Library Documentation

Overview

The Rock 7 **RockBLOCK** is a fascinating new communications module that gives TTL-level devices like Arduino access to the Iridium satellite network. This is a big deal, because it means that your application can now easily and inexpensively communicate from any point on the surface of the globe, from the heart of the Amazon to the Siberian tundra.

This library, **IridiumSBD**, uses the RockBLOCK/Iridium's **SBD** ("Short Burst Data") protocol to send and receive short messages to/from the Iridium hub. SBD is a "text message"-like technology that supports the transmission of text or binary messages up to a certain maximum size (270 bytes received, 340 bytes transmitted).

For more information, visit the Rock 7 website.

"3-Wire" Wiring

Rock 7 have made interfacing to the Arduino quite simple. Each RockBLOCK ships with a 10-pin JST-terminated cable that snaps onto the RockBLOCK and conveniently exposes the various signal lines for the client device to connect to. It's actually only necessary to connect 4 or 5 of these to get your Arduino application up and running. In our configuration we ignore the flow control lines and talk to the RockBLOCK over what Iridium calls a "3-wire" TTL serial interface. In wiring table below, we assume that the RockBLOCK is being powered from the Arduino 5V power bus.

RockBLOCK Connection	Arduino Connection
+5V (Power)	+5V (Power)
GND	GND
TX	TX Serial Pin*
RX	RX Serial Pin*
SLEEP	+5V or GPIO pin*

A minimal "3-wire" connection to RockBLOCK

*The TX and RX lines are labeled on the RockBLOCK as viewed *from the Arduino*, so the TX line would be transmitting serial data to the RockBLOCK. These lines support TTL-level serial (default 19200 baud), so you can either connect it directly to the/a built-in UART or create a "soft" serial on any two suitable pins. We usually opt for the latter choice to free up the UART(s) for diagnostic and other console communications. The active low SLEEP wire may be connected to a 5V line (indicating that the device is perpetually awake), but it's a good power saving technique to connect it to a general-purpose pin, allowing the library to put the RockBLOCK in a low power "sleep" state when its services are not needed.

Non-blocking Retry Strategy

The nature of satellite communications is such that it often takes quite a long time to establish a link. Satellite communications are line-of-sight, so having a clear view of an unclouded sky greatly improves speed and reliability; however, establishing contact may be difficult even under ideal conditions for the simple reason that at a given time no satellite may immediately be overhead. In these cases, the library initiates a moderately elaborate behind-the-scenes series of retries, waiting for a satellite to appear.

With a clear sky, transmissions almost always succeed after a few of these retries, but the entire process may take up to several minutes. Since most microcontroller applications cannot tolerate blocking delays of this length, **IridiumSBD** provides a callback mechanism to ensure that the Arduino can continue performing critical tasks. That is, if the library user provides a global C++ function with the signature

```
bool ISBDCallback();
```

(and this is highly recommended for all but the most trivial applications), then that function will be called repeatedly while the library is waiting for long operations to complete. In it you can take care of activities that need doing while you're waiting for the transmission to complete. Simple example:

```
bool ISBDCallback()
{
   unsigned ledOn = (bool)((millis() / 1000) % 2);
   digitalWrite(ledPin, ledOn); // Blink LED every second
   return true;
}

...
// This transmission may take a long time, but the LED keeps blinking
isbd.sendSBDText("Hello, mother!");
```

Note: It is not permitted to call most IridiumSBD methods from within the callback. Doing so will immediately return **ISBD_REENTRANT.**
Note: Your application can prematurely terminate a pending IridiumSBD operation by returning **false** from the callback. Doing so causes the pending operation to return **ISBD_CANCELLED**.

Power Considerations

The RockBLOCK module uses a "super capacitor" to supply power to the Iridium 9602. As the capacitor is depleted through repeated transmission retries, the host device's power bus replenishes it. Under certain low power conditions it is important that the library not retry too quickly, as this can drain the capacitor and render the 9602 inoperative. In particular, when powered by a low-power 90 mA max USB supply, the interval between transmit retries should be extended to as much as 60 seconds, compared to 20 for, say, a high-current battery solution.

To transparently support these varying power profiles, **IridiumSBD** provides the ability to fine-tune the delay between retries. This is done by calling

```
isbd.setPowerProfile(1); // For USB "low current" applications
```

or

```
isbd.setPowerProfile(0); // For “high current” applications
```

Construction and Startup

To begin using the library, first create an **IridiumSBD** object. The **IridiumSBD** constructor binds the new object to an Arduino **Stream** (i.e. the RockBLOCK serial port) and, optionally, the RockBlock SLEEP line:

```
IridiumSBD(Stream &stream, int sleepPinNo = -1);
```

Example startup:

```
#include "IridiumSBD.h"
#include "SoftwareSerial.h"

SoftwareSerial ssIridium(18, 19); // RockBLOCK serial port on 18/19
IridiumSBD isbd(ssIridium, 10);   // RockBLOCK SLEEP pin on 10

void setup()
{
   isbd.setPowerProfile(1); // This is a low power application
   isbd.begin(); // Wake up the 9602 and prepare it for communications.
   ...
```

Data transmission

The methods that make up the meat of the **IridiumSBD** public interface, are, naturally, those that enable the sending and receiving of data. There are four such functions in **IridiumSBD**, two “send-only” functions (text and binary), and two “send-and-receive” functions (again, text and binary):

```
// Send a text message
int sendSBDText(const char *message);

// Send a binary message
int sendSBDBinary(const uint8_t *txData, size_t txDataSize);

// Send a text message and receive one (if available)
int sendReceiveSBDText(const char *message, uint8_t *rxBuffer,
       size_t &rxBufferSize);

// Send a binary message and receive one (if available)
int sendReceiveSBDBinary(const uint8_t *txData, size_t txDataSize,
       uint8_t *rxBuffer, size_t &rxBufferSize);
```

Send-only and Receive-only applications

Note that at the lowest-level, SBD transactions always involve the sending *and* receiving of exactly one message (if one is available). That means that if you call the simpler variants **sendSBDText** or **sendSBDBinary** and there happen to messages in your incoming (RX) message queue, the first of these is discarded and irrevocably lost. This may be perfectly acceptable for a “send-only” application that never expects to receive messages, but if there is some chance that you will need to process an inbound message, call **sendReceiveSBDText** or **sendReceiveSBDBinary** instead.

If your application is *receive-only*, simply call **sendReceiveSBDText** with a NULL outbound **message** parameter.

If no inbound message is available, the **sendReceive*** messages indicate this by returning **ISBD_SUCCESS** and setting **rxBufferSize** to 0.

Diagnostics

IridiumSBD provides two methods for performing self-diagnostics. These are

```
void attachConsole(Stream &stream);
void attachDiags(Stream &stream);
```

These allow the host application to provide a Stream object (serial port) that can be used to monitor the RockBLOCK serial traffic and diagnostic messages, respectively. The typical usage is to simply use the Arduino serial port to monitor both of these—assuming that it is connected to a PC serial console and not otherwise used:

```
isbd.attachConsole(Serial);
isbd.attachDiags(Serial);
```

Receiving Multiple Messages

After every successful SBD send/receive operation, the Iridium satellite system informs the client how many messages remain in the inbound message queue. The library reports this value with the **getWaitingMessageCount** method. Here's an example of a loop that reads all the messages in the inbound message queue:

```
do
{
   char rxBuffer[100];
   size_t bufferSize = sizeof(rxBuffer);
   int status = isbd.sendReceiveText("Is anyone home?”,
      rxBuffer, bufferSize);
   if (status != ISBD_SUCCESS)
   {
      /* ...process error here... */
      break;
   }
   if (bufferSize == 0)
      break; // all done!
   /* ...process message in rxBuffer here... */
} while (isbd.getWaitingMessageCount() > 0);
```

Erratum Workarounds and other Tweaks

In May, 2013, Iridium identified a potential problem that could cause a satellite modem like the RockBLOCK to lock up unexpectedly. In a product bulletin they stated that though future versions of the Iridium firmware would resolve this issue, current software should work around it by adding an

"MSSTM" software query to the device. **IridiumSBD** employs this important workaround by default, but you can disable it with:

```
isbd.useMSSTMWorkaround(false);
```

Similarly, Iridium recommend that transmission be deferred until the signal quality has reached at least 2 (on a scale of 0 to 5, as reported by getSignalQuality()). However, there are times when it is advantageous to use a lower (or higher) value than 2 as this default minimum. To change the default minimum from 2, use:

```
isbd.setMinimumSignalQuality(1);
```

Error return codes

Many **IridiumSBD** methods return an integer error status code, with ISBD_SUCCESS (0) indicating successful completion. These include **begin**, **sendSBDText**, **sendSBDBinary**, **sendReceiveSBDText**, **sendReceiveSBDBinary**, **getSignalQuality**, and **sleep**. Here is a complete list of the possible error return codes:

```
#define ISBD_SUCCESS             0
#define ISBD_ALREADY_AWAKE       1
#define ISBD_SERIAL_FAILURE      2
#define ISBD_PROTOCOL_ERROR      3
#define ISBD_CANCELLED           4
#define ISBD_NO_MODEM_DETECTED   5
#define ISBD_SBDIX_FATAL_ERROR   6
#define ISBD_SENDRECEIVE_TIMEOUT 7
#define ISBD_RX_OVERFLOW         8
#define ISBD_REENTRANT           9
#define ISBD_IS_ASLEEP           10
#define ISBD_NO_SLEEP_PIN        11
```

Interface Documentation

`IridiumSBD(Stream &stream, int sleepPinNo = -1);`

Description: Creates an IridiumSBD library object
Returns: N/A [constructor]
Parameter: **stream** - The serial port that the RockBLOCK is connected to.
Parameter: **sleepPin** - The number of the Arduino pin connected to the RockBLOCK SLEEP line.

Note: Connecting and using the sleepPin is recommended for battery-based solutions. Use **sleep**() to put the RockBLOCK into a low-power state, and **begin**() to wake it back up.

`int begin();`

Description: Starts (or wakes) the RockBLOCK modem.
Returns: ISBD_SUCCESS if successful, a non-zero code otherwise.
Parameter: None.

Notes

- **begin**() also serves as the way to wake a RockBLOCK that is asleep.
- At initial power up, this method make take several tens of seconds as the device charges. When waking from sleep the process should be faster.
- If provided, the user's **ISBDCallback** function is repeatedly called during this operation.
- This function should be called before any transmit/receive message

`int sendSBDText(const char *message);`

Description:	Transmits a text message to the global satellite system.
Returns:	ISBD_SUCCESS if successful, a non-zero code otherwise;
Parameter:	**message** – A 0-terminated string message.

Notes

- The library calculates retries the operation for up to 300 seconds by default. (To change this value, call **adjustSendReceiveTimeout**.)
- The maximum size of a transmitted packet (including header and checksum) is 340 bytes.
- If there are any messages in the RX queue, the first of these is discarded when this function is called.
- If provided, the user's **ISBDCallback** function is repeatedly called during this operation.

`int sendSBDBinary(const uint8_t *txData, size_t txDataSize);`

Description:	Transmits a binary message to the global satellite system.
Returns:	ISBD_SUCCESS if successful, a non-zero code otherwise;
Parameter:	**txData** – The buffer containing the binary data to be transmitted.
Parameter:	**txDataSize** - The size of the buffer in bytes.

Notes

- The library calculates and transmits the required headers and checksums and retries the operation for up to 300 seconds by default. (To change this value, call **adjustSendReceiveTimeout**.)
- The maximum size of a transmitted packet (including header and checksum) is 340 bytes.
- If there are any messages in the RX queue, the first of these is discarded when this function is called.
- If provided, the user's **ISBDCallback** function is repeatedly called during this operation.

`int sendReceiveSBDText(const char *message, uint8_t *rxBuffer, size_t &rxBufferSize);`

Description:	Transmits a text message to the global satellite system and receives a message if one is available.
Returns:	ISBD_SUCCESS if successful, a non-zero code otherwise;
Parameter:	**message** – A 0-terminated string message.
Parameter:	**rxBuffer** – The buffer to receive the inbound message.
Parameter:	**rxBufferSize** - The size of the buffer in bytes.

Notes

- The library calculates retries the operation for up to 300 seconds by default. (To change this value, call **adjustSendReceiveTimeout**.)
- The maximum size of a transmitted packet (including header and checksum) is 340 bytes.
- The maximum size of a received packet is 270 bytes.
- If there are any messages in the RX queue, the first of these is discarded when this function is called.
- If provided, the user's **ISBDCallback** function is repeatedly called during this operation.
- The library returns the size of the buffer actually received into **rxBufferSize**. This value should always be set to the actual buffer size before calling **sendReceiveSBDText**.

`int sendReceiveSBDBinary(const uint8_t *txData, size_t txDataSize, uint8_t *rxBuffer, size_t &rxBufferSize);`

Description: Transmits a binary message to the global satellite system and receives a message if one is available.
Returns: ISBD_SUCCESS if successful, a non-zero code otherwise;
Parameter: **txData** – The buffer containing the binary data to be transmitted.
Parameter: **txDataSize** - The size of the outbound buffer in bytes.
Parameter: **rxBuffer** – The buffer to receive the inbound message.
Parameter: **rxBufferSize** - The size of the buffer in bytes.

Notes

- The library calculates and transmits the required headers and checksums and retries the operation for up to 300 seconds by default. (To change this value, call **adjustSendReceiveTimeout**.)
- The maximum size of a transmitted packet (including header and checksum) is 340 bytes.
- The maximum size of a received packet is 270 bytes.
- If there are any messages in the RX queue, the first of these is discarded when this function is called.
- If provided, the user's **ISBDCallback** function is repeatedly called during this operation.

`int sendReceiveSBDBinary(const uint8_t *txData, size_t txDataSize, uint8_t *rxBuffer, size_t &rxBufferSize);`

Description: Transmits a binary message to the global satellite system and receives a message if one is available.
Returns: ISBD_SUCCESS if successful, a non-zero code otherwise;
Parameter: **txData** – The buffer containing the binary data to be transmitted.
Parameter: **txDataSize** - The size of the outbound buffer in bytes.
Parameter: **rxBuffer** – The buffer to receive the inbound message.
Parameter: **rxBufferSize** - The size of the buffer in bytes.

Notes

- The library calculates and transmits the required headers and checksums and retries the operation for up to 300 seconds by default. (To change this value, call **adjustSendReceiveTimeout**.)
- The maximum size of a transmitted packet (including header and checksum) is 340 bytes.
- The maximum size of a received packet is 270 bytes.

- If there are any messages in the RX queue, the first of these is discarded when this function is called.
- If provided, the user's **ISBDCallback** function is repeatedly called during this operation.

`int getSignalQuality(int &quality);`

Description:	Queries the signal strength and visibility of satellites
Returns:	ISBD_SUCCESS if successful, a non-zero code otherwise;
Parameter:	**quality** – Return value: the strength of the signal (0=nonexistent, 5=high)

Notes

- If provided, the user's **ISBDCallback** function is repeatedly called during this operation.
- This method is mostly informational. It is not strictly necessary for the user application to verify that a signal exists before calling one of the transmission functions, as these check signal quality themselves.

`int getWaitingMessageCount();`

Description:	Returns the number of waiting messages on the Iridium servers.
Returns:	The number of messages waiting.
Parameter:	None.

Notes

- This number is only valid if one of the send[Receive] methods have previously completed successfully. If not, the value returned from **getWaitingMessageCount** is -1 ("unknown").

`int sleep();`

Description:	Puts the RockBLOCK into low power "sleep" mode
Returns:	ISBD_SUCCESS if successful, a non-zero code otherwise;
Parameter:	**None.**

Notes

- This method gracefully shuts down the RockBLOCK and puts it into low-power standby mode by bringing the active low SLEEP line low.
- Wake the device by calling **begin**.
- If provided, the user's **ISBDCallback** function is repeatedly called during this operation.

`bool isAsleep();`

Description:	indicates whether the RockBLOCK is in low-power standby mode.
Returns:	**true** if the device is asleep
Parameter:	**None.**

`void setPowerProfile(int profileNo);`

Description:	Defines the device power profile
Returns:	**None**.
Parameter:	**profileNo** – 1 for low-current USB power source, 0 for default power

Notes

- This method defines the internal delays between retransmission. Low current applications need longer delays.

`void adjustATTimeout(int seconds);`

Description: Adjusts the internal timeout timer for serial AT commands

Returns: None.

Parameter: **seconds** – The maximum number of seconds to wait for a response to an AT command (default=20).

Notes

- The Iridium 9602 frequently does not respond immediately to an AT command. This value indicates the number of seconds IridiumSBD should wait before giving up.
- It is not expected that this method will be commonly used.

`void adjustSendReceiveTimeout(int seconds);`

Description: Adjusts the internal timeout timer for the library send[Receive] commands

Returns: None.

Parameter: **seconds** – The maximum number of seconds to continue attempting retransmission of messages (default=300).

Notes

- This setting indicates how long IridiumSBD will continue to attempt to communicate with the satellite array before giving up. The default value of 300 seconds (5 minutes) seems to be a reasonable choice for many applications, but higher values might be more appropriate for others.

`void setMinimumSignalQuality(int quality);`

Description: Defines the minimum signal quality needed to begin a send/receive transmission.

Returns: None.

Parameter: **quality** – The minimum signal quality on a scale of 0 (nonexistent) to 5 (superb) needed before the library allows a transmission to begin. (default=2)

Notes

- Iridium recommend using 2 for this value, although there are occasions where 1 might be used.

`void useMSSTMWorkaround(bool useWorkaround);`

Description: Defines whether the library should use the technique described in the Iridium Product Advisor of 13 May 2013 to avoid possible lockout.

Returns: None.

Parameter: **useWorkaround** – "true" if the workaround should be employed; false otherwise. This value is set internally to "true" by default, on the assumption that the attached device may have an older firmware.

Notes

- Affected firmware versions include TA11002 and TA12003. If your firmware version is later than these, you can save some time by setting this value to false.

`void attachConsole(Stream &stream)`

Description: Binds **stream** as the library output console.
Returns: **None**.
Parameter: **stream** – The stream object (typically a serial port) that will monitor the serial traffic to/from the RockBLOCK device.

Notes

- This is a diagnostic routine. Use it to monitor traffic on a PC console.

`void attachDiags(Stream &stream)`

Description: Binds **stream** as the library diagnostic stream
Returns: **None**.
Parameter: **stream** – The stream object (typically a serial port) that will display library diagnostic messages.

Notes

- This is a diagnostic routine. Use it to monitor debug messages on a PC console.
- If this functionality is no longer needed, you can recover system resources by recompiling the library source without diagnostics. Change the line near the top of IridiumSBD.h to

```
#define ISBD_DIAGS            0
```

`bool ISBDCallback()`

Description: This is not a library method, but an (optional) user-provided callback
Returns: **true** if the operation should continue, **false** to terminate it.
Parameter: **None.**

Notes

- If this function is not provided the library methods will block.

License

This library is distributed under the terms of the GNU LGPL license.

Download

The latest revision of the library is 0.2.

Revision history

0.1 - Initial draft submitted to Rock 7 for review

0.2 – Added text about the AT-MSSTM erratum/workaround and changing the minimum required signal quality. Also documented related new methods setMinimumSignalQuality() and useMSSTMWorkaround().