PaulZC wrote: "This guide does not yet include the new I^{2}C functionality..."

IridiumSBD Arduino Library rev 1.0

Overview

The Rock 7 **RockBLOCK** is a fascinating 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 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 conveniently exposes many signal lines for the client device, but it's actually only necessary to connect 4 or 5 of these to get your 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 (optional)	+5V or GPIO pin**
RING (optional)	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 a built-in UART or create a "soft" serial on any two suitable pins. We usually opt for the latter on smaller devices like Uno 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. The RING line is used to alert the client that a message is available.

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 satellites are not always overhead. In these cases, the library initiates a behind-the-scenes series of retries, waiting for satellites to appear.

With a clear sky, transmissions eventually almost always succeed, 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. Specifically, if the library user provides a global C++ function with the signature

```
bool ISBDCallback();
```

(and this is 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. As a simple example, this blinks an LED during the library operations:

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

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

Note: Most IridiumSBD methods are not available from within the callback and return the error code **ISBD_REENTRANT** when called.
Note: Your application can prematurely terminate a pending IridiumSBD operation by returning **false** from the callback. This causes the pending operation to immediately return **ISBD_CANCELLED**.

Power Considerations

The RockBLOCK module uses a "super capacitor" to supply power to the Iridium 9602/9603. As the capacitor is depleted through repeated transmission attempts, 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 device 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

```
// For USB “low current” applications
```

```
modem.setPowerProfile(IridiumSBD::USB_POWER_PROFILE);
```

or

```
// For "high current" (battery-powered) applications
modem.setPowerProfile(IridiumSBD::DEFAULT_POWER_PROFILE);
```

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 device's serial port) and, optionally, its SLEEP and RING lines:

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

Example startup:

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

SoftwareSerial ssIridium(18, 19); // RockBLOCK serial port on 18/19
IridiumSBD modem(ssIridium, 10);  // SLEEP pin on 10, RING pin not connected

void setup()
{
   modem.setPowerProfile(IridiumSBD::USB_POWER_PROFILE);
   modem.begin(); // Wake up the 9602 and prepare it for communications.
   ...
```

Data transmission

The methods that make up the core of the **IridiumSBD** public interface, are, naturally, those that send and receive data. There are four such methods 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, Iridium SBD transactions always involve the sending *and* receiving of exactly one message (if one is available). That means that if you call the send-only variants **sendSBDText** or **sendSBDBinary** and messages happen to be waiting in your incoming (RX) message queue, the first of these is discarded and irrevocably lost. This may be perfectly acceptable for an application that doesn't care about inbound messages, but if there is some chance that you will need to process one, call **sendReceiveSBDText** or **sendReceiveSBDBinary** instead.

If your application is *receive-only*, 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 operates by maintaining a TTL-serial dialog with the Iridium 9602/3. To diagnose failures it is often useful to "spy" on this conversation. If you provide a callback function with the signature

void ISBDConsoleCallback(IridiumSBD *device, char c);

the library will call it repeatedly with data in this conversation, and your application can log it. Similarly, to inspect the run state of the library as it is working, simply provide a callback like this:

void ISBDDiagsCallback(IridiumSBD *device, char c);

These callbacks allow the host application to monitor Iridium serial traffic and the library's diagnostic messages. The typical usage is to simply forward both to the Arduino serial port for display on a computer terminal:

```
void ISBDConsoleCallback(IridiumSBD *device, char c)
{
  Serial.write(c);
}

void ISBDDiagsCallback(IridiumSBD *device, char c)
{
  Serial.write(c);
}
```

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 = modem.sendReceiveText(NULL, rxBuffer, bufferSize);
   if (status != ISBD_SUCCESS)
   {
      /* ...process error here... */
      break;
   }
```

```
      if (bufferSize == 0)
         break; // all done!
      /* ...process message in rxBuffer here... */
   } while (modem.getWaitingMessageCount() > 0);
```

*Note that **getWaitingMessageCount** is only valid after a successful send/receive operation.

Erratum Workaround

In May, 2013, Iridium identified a potential problem that could cause a satellite modem like the RockBLOCK to lock up unexpectedly. This issue only affects devices with firmware older than version TA13001: use getFirmwareVersion() to check yours. The library automatically employs the workaround recommended by Iridium — a successful check of the system time with AT-MSSTM before each transmission — by default for firmware that is older, but you can disable this check with:

```
modem.useMSSTMWorkaround(false);
```

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
#define ISBD_NO_NETWORK          12
#define ISBD_MSG_TOO_LONG        13
```

IridiumSBD Interface

Constructor

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

Description:	Creates an IridiumSBD library object
Returns:	N/A
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.

Startup

```
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 operation

Data transmission

```
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.

Utilities

```
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, or -1 if unknown.
Parameter:	None.

Notes

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

```
int getSystemTime(struct tm &tm)
```

Description: Returns the system time from the Iridium network.
Returns: ISBD_SUCCESS if successful, a non-zero code otherwise.
Parameter: **tm** – the time structure to be filled in

Notes

- This method returns the Iridium network time in tm.
- “tm” is a C standard library structure defined in <time.h>
- Note that the tm_mon field is zero-based, i.e. January is 0
- This function uses AT+MSSTM, which might report “Network not found”. In this case, the function returns ISBD_NO_NETWORK.

```
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.**

```
bool isRingAsserted()
```

Description: indicates whether the RockBLOCK’s RING line is asserted
Returns: **true** if RING is asserted
Parameter: **None.**

Notes

- The Iridium documentation says that RING is asserted for 5 seconds after an incoming message is detected, or until a message transfer takes place.

```
int getFirmwareVersion(char *version, size_t bufferSize)
```

Description:	Returns a string representing the firmware revision number.
Returns:	ISBD_SUCCESS if successful, a non-zero code otherwise.
Parameter:	**version** – the buffer to contain the version string
Parameter:	**bufferSize** – the size of the buffer to be filled

Notes

- This method returns the version string in the **version** buffer.
- bufferSize should be at least 8 to contain strings like TA13001 with the 0 terminator.

```
void setPowerProfile(POWERPROFILE profile)
```

Description:	Defines the device power profile
Returns:	**None**.
Parameter:	**profile** – USB_POWER_PROFILE for low-current USB power source, DEFAULT_POWER_PROFILE for default (battery) power

Notes

- This method defines the internal delays between retransmission. Low current applications may require 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 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 enableRingAlerts(bool enable)`

Description: Overrides whether the library should enable the RING alert signal pin and the unsolicited SBDRING notification.
Returns: None.
Parameter: **enable** – "true" if RING alerts should be enabled.

Notes

- This method uses the Iridium AT+SBDMTA to enable or disable alerts.
- This method take effect at the next call to **begin()**, so typically you would call this before you start your device.
- RING alerts are enabled by default if the library user has specified a RING pin for the IridiumSBD constructor. Otherwise they are disabled by default. Use this method to override that as needed.

Callbacks (optional)

`bool ISBDCallback()`

Description: An optional user-supplied callback to help provide the appearance of responsiveness during lengthy Iridium operations
Returns: **true** if the calling library operation should continue, **false** to terminate it.
Parameter: **None.**

Notes

- If this function is not provided the library methods will appear to block.
- This is not a library method, but an optional user-provided callback function.

```
void ISBDConsoleCallback(IridiumSBD *device, char c)
```

Description:	An optional user-supplied callback to sniff the conversation with the Iridium 9602/3.
Returns:	**None.**
Parameter:	**device** – a handle to the modem device
Parameter:	**c** – a character in the conversation

Notes
- Typical usage is to write **c** to a console for diagnostics.
- This is not a library method, but an optional user-provided callback function.

```
void ISBDDiagsCallback(IridiumSBD *device, char c)
```

Description:	An optional user-supplied callback to monitor the library's run state.
Returns:	**None.**
Parameter:	**device** – a handle to the modem device
Parameter:	**c** – a character in the run log

Notes
- Typical usage is to write **c** to a console for diagnostics
- This is not a library method, but an optional user-provided callback function.

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

Download
The latest revision of the library is 2.0, available at http://github.com/mikalhart/IridiumSBD.git.

Document revision history

Version	Date	Author	Reason
1.0	2013	Mikal Hart	Initial draft submitted to Rock 7 for review
1.1	2014	Mikal Hart	Added text about the AT-MSSTM erratum/workaround and changing the minimum required signal quality. Also documented related new methods **setMinimumSignalQuality** and **useMSSTMWorkaround**().
2.0	21 October 2017	Mikal Hart	Several API revisions. Removed **setMinimumSignalQuality** (no longer used), added support for RING monitoring (**enableRingAlerts** method, and new RING alert pin on constructor), and changed the way diagnostics are done by replacing the **attachConsole** and **attachDiags** methods with user-supplied callbacks

			ISBDConsoleCallback and **ISBDDiagsCallback**. Added **getSystemTime** and **getFirmwareVersion** utility functions. Add explanation that MSSTM workaround is no longer enabled by default if firmware is sufficiently new (TA13001 or newer).