Qualcomm Technologies International, Ltd.

Implementing Streams in BlueCore Applications

User Guide

80-CT437-1 Rev. AN

January 28, 2018

Revision history

Revision	Date	Description
1	JUL 2010	Original publication of this document. Alternative document number CS-00207483-UG.
2	MAR 2011	Updated to latest style.
3	JUL 2011	Technical corrections
4	JUL 2011	Correction to Table A.1
5	OCT 2011	Correction to description of the `TransformSlice` function.
6	JAN 2012	Updated to latest style
7	DEC 2012	Updated to match latest code implementations.
8	APR 2014	Updated to latest style
9	MAY 2015	Technical updates
10	SEP 2016	Updated to conform to QTI standards; No technical content was changed in this document revision
11	APR 2017	Formatting changes for source code examples. Restructured the content. Removed API documentation already provided in the SDK API documentation.
AM	OCT 2017	Added to Content Management System. DRN updated to use the Agile number. No technical content was changed in this document revision
AN	JAN 2018	Added information on Pipe stream support

Contents

Tables

Figures

1 Implementing streams in BlueCore - Overview

Streams provide an efficient method of controlling data flow within chip applications. They are used to transfer data over the air between connected Bluetooth devices, between processors in the device, or internally on the IC.

The QTIL Bluetooth SDK libraries provide functions for efficiently managing streams.

2 What is a stream?

A stream is a logical construct used to conceptualize a data endpoint on the BlueCore, that is the UART, a port on the Qualcomm® Kalimba™ DSP, or an audio codec for example. Generally, a stream describes both a data source and sink.

For example, the UART can send and receive data, and therefore contains both a source and a sink.

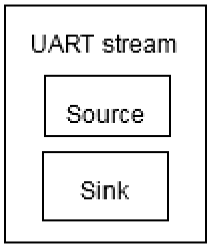

Figure 2-1 UART stream example

In practice, BlueCore applications do not deal with any stream objects directly. Instead, they create source and sink objects, and control data flow by connecting together sources and sinks, generally from different streams. However, in this document and elsewhere, the term stream may describe either a source or a sink.

QTIL Bluetooth SDKs provide developers with functions to obtain, configure and connect sinks and sources.

NOTE It is a common misconception that a stream describes the flow of data from a source to a sink. In fact, the data flow is usually described as a Transform. See Managed connections between sinks and sources

2.1 Overview of sinks and sources

Data can be written to a sink and read from a source. An application may:

- Read data from a source and interpret it directly
- Generate data and write it to a sink
- Read data from a source, process it, then write it to a sink

or

- Connect a source and sink together, so that data arriving at one is automatically transferred to the other.

There can be several sources and sinks in use by an application at the same time. However, only one source and one sink can be mapped into the address space and therefore accessed by the application at any time.

2.2 Connecting sinks and sources

For data to flow between a sink and a source, they must be connected. Bluetooth SDKs support two methods of connecting sinks and sources.

NOTE (1) Generally, the sink and source come from different streams, but it is also possible to connect the source of one stream to the sink of the same stream.

(2) It is possible to duplicate a stream so that one source can be streamed to two sinks, see Common source and sink functionality

2.2.1 Direct connections

Data is transferred automatically from the source to the sink. Data flows consistently and is not visible to the application. Due to the way the physical hardware behaves, there are restrictions on which types of source can be connected to which types of sink using direct connections.

Such connections are implanted as transform, manipulating the data appearing at the source, and automatically routing it to the sink. The most basic transform is a simple copy and is what is usually created when streams are connected using `StreamConnect()`. However, it is also possible for a variety of modifications to be made to the data as it is moved around.

Do not confuse the role of a stream in a transform with the source and sink belonging to an individual stream. Most streams have both a source and a sink, but take only one role when connected to a transform. See Direct connections between sinks and sources

2.2.2 Managed connections

Managed connections give the application visibility of the data held in sinks and sources, and allows data transfer to be controlled by the application. This can be useful for managing scenarios involving data stalls, but reduces speed and increases complexity.

There are also some restrictions on which sources and sinks can be connected by a managed connection. See Managed connections between sinks and sources

2.3 Stream types

The QTIL Bluetooth SDKs distinguish the following stream types:

- RFCOMM: Serial Cable Emulation Protocol
- L2CAP: Logical Link Control and Adaptation Protocol
- SCO: Synchronous Connection Oriented link
- Host (BCSP): BlueCore Serial Protocol
- Serial port (raw UART)
- Kalimba DSP port
- USB: Universal Serial Bus
- Audio
- Region (an area of application memory, acting as a source only)

- Files (acting as a source only)
- Ringtone (acting as a source only)
- I2C (acting as source only)
- ATT (streams from a specific handle or handles in an ATT Database of a remote device, acting as source only)
- FastPipe
- Flash Partition
- Pipe

Most streams may act as both a `source` and a `sink`, but there are some exceptions (as noted above). Not all stream types may be available with all products.

2.4 Examples of stream connections

Figure 2-2 to Figure 2-5 show examples of how streams may be connected together to create a variety of typical stream implementations. The arrows indicate the flow of data.

Managed connection: UART to application

Figure 2-2 shows a situation where the application reads data from the UART, the data can then be processed before being returned to the UART.

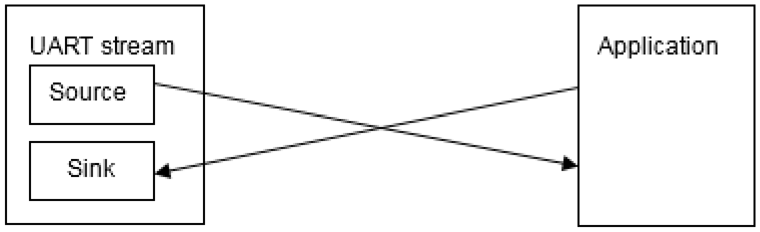

Figure 2-2 Reading from UART

Managed connections: file to application and application to UART

The application reads data from a file, the data is processed, before being sent to the UART.

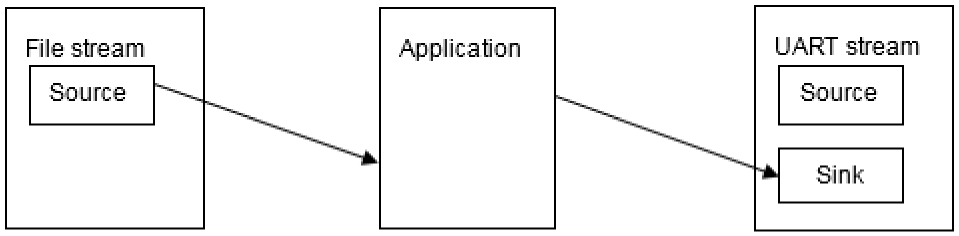

Figure 2-3 Reading from file, routing to UART

Direct connection: audio subsystem to DSP

The audio stream is connected to the DSP, which processes and returns the data to the audio subsystem e.g. connecting an audio ADC/DAC pair to a DSP port.

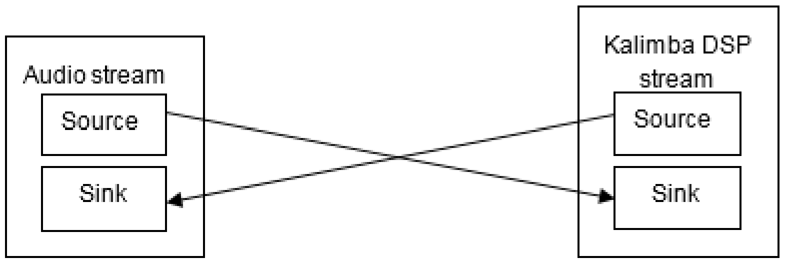

Figure 2-4 Direct connection

Simple UART loopback

Figure 2-5 shows a direct connection between the UART stream’s sink and its source. Any data arriving at the UART sink is automatically forwarded to the source creating a loopback.

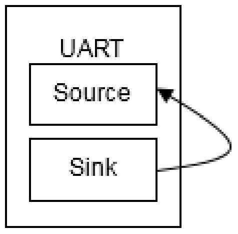

Figure 2-5 UART loopback

3 Overview of the source and sink API

Functions declared in the `sink.h`, `source.h`, and `stream.h` header files are used to obtain sinks and sources that can then be connected together to stream data.

The functions to obtain sinks and sources can be considered in pairs, one of which obtains a sink and one a source. In most cases, streams are bi-directional and so the `StreamConnect` function is called twice when setting up a stream.

For example:

Figure 3-1 StreamConnect example

```
/* Connect the UART to DSP Port 3 */
PanicNull(StreamConnect(StreamKalimbaSource(3), StreamUartSink()));
PanicNull(StreamConnect(StreamUartSource(), StreamKalimbaSink(3)));
```

Functions that return sources and sinks such as `StreamUartSource()` and `StreamRfcommSink()` use the special return value of zero to indicate failure. This is possible because both the source and sink types are actually opaque pointers, that is, the pointer can be used to manipulate the source or sink without needing to know its structure. The importance of returning zero on failure is that it allows the code to check for failure using the `PanicNull` function.

Resource management

Obtaining a source or a sink is not, in general, free. Even if they are not connected to anything by a Transform, some sources and sinks reserve resources in the underlying system at the point they are created. This restricts the type and number of streams that can be created at the same time.

For example, PCM, SPDIF and I^{2}S (three types of audio stream) use a shared bus in hardware and so are mutually exclusive. They also reserve audio channels for their own use, restricting the number of sources or sinks that can be created of the same type (for example, only four PCM slots exist). In general, the actual restrictions depend on the specific chip and firmware used.

Applications should be aware that requests for sources and sinks may fail due to these constraints, and close any that are no longer necessary, see Functions to close a source and sink.

3.1 Fuctions to obtain sources and sinks

In general, the function prototypes for obtaining a source or a sink is as follows:

Figure 3-2 Functions for obtaining sources and sinks

```
Source Stream<x>Source(<type> <identifier>)
Sink Stream<x>Sink(<type> <identifier>)
```

Where `x` is the kind of stream and type and identifier are uses to uniquely identify the source or sink. For example, here are the function prototypes for obtaining a source or sink for an RFCOMM stream:

Figure 3-3 Example of obtaining RFCOMM source and sink

```
Source StreamRfcommSource (uint16 conn_id);
Sink StreamRfcommSink (uint16 conn_id);
```

The `conn_id` is provided by BlueStack to uniquely identify and reference the RFCOMM connection. If the source or sink cannot be created, then these functions return `0`. This could be because the identifier is not correct or because that stream type is not enabled or available.

NOTE It is uncommon to see such function calls in applications because the Connection library, which handles creating connections between devices, generates `CFM` messages that return the sink associated with the connection.

3.2 Functions to close a source and sink

Description of how to close a source or sink and ensure that their resources are successfully freed:

Figure 3-4 Example of closing a source and sink

```
bool SourceClose(Source source);
bool SinkClose(Sink sink);
```

If the source or sink was successfully closed, these functions return `TRUE`, otherwise `FALSE`.

NOTE A source or sink stream can only be closed if it is not connected to any other source or sink. Therefore, a `TransformDisconnect()` or `StreamDisconnect()` function call should be made before the `SourceClose()` or `SinkClose()` function call. Some sources, such as RFCOMM connections or the USB hardware, have a lifetime defined by other means, and cannot be closed using a call to `SourceClose()`.

4 Direct connections between sinks and sources

Directly connected sources and sinks are set up by the application for example, the source or sink connection is created and connected by the application, but once connected by a transform, the dataflow is entirely controlled by the Firmware. Generally, before a source or sink can be closed, the stream or transform must be disconnected.

Refer to the `stream.h` and `transform.h` API documentation for detailed information on the available functionality.

General

The simplest way of directly connecting a source and sink is using the `StreamConnect()` function, which copies data directly from the source to the sink. This data transfer is handled by the firmware, and so it is efficient and fast. However, the data being streamed is not visible to the application (unless the stream from source to sink is disconnected using `StreamDisconnect()`).

The `StreamConnect()` function creates the simplest form of transform and returns a transform type with the transform already in progress (it does not need to be started).

Some sources must have their buffers emptied before they can closed. The `StreamDispose()` function does this. However, the `StreamConnectAndDispose()` function provides the most robust behavior:

- Provides a straightforward copy transform like `StreamConnect()`.
- If the connection is broken by `StreamDisconnect()` or the sink is closed independently, the source is passed to `StreamDispose()` so that the source is emptied correctly.

See the `stream.h` API documentation for more details on these and other stream functions.

The copy transform created by the `StreamConnect()` and `StreamConnectAndDispose()` functions is the simplest type of transform. Other types of transform can be created and are described in more detail in Transform creation functions.

Direct connection stream compatibility describes functions that can be used to control and configure those transforms, when they have been created.

4.1 StreamConnect function

`StreamConnect` and related functions are declared in the `stream.h` header file.

prototype

- `Transform StreamConnect (Source source, Sink sink)`

Parameters

- `source`: The source to be connected.
- `sink`: The sink to be connected.

Description

`StreamConnect` establishes an automatic connection between the specified source and sink. Data arriving at the source is automatically copied to the sink when space permits. This is the most efficient but least flexible option available for connecting streams.

The function returns a Transform (which is now in progress) if the connection was successfully established, or zero if the operation failed. For more details on Transforms see Transform creation functions and Transform control functions.

To use this function, the source and sink must be valid and cannot be involved in an existing connection. The connection is broken if either the source or sink become invalid.

When a source and sink are connected using `StreamConnect`, operations such as `SinkClaim`, `SourceClaim`, `SinkMap`, and so forth. are refused. Therefore, to insert data into a connected sink it is necessary to disconnect the stream, claim the sink, write and flush the data being inserted and then reconnect the stream using `StreamConnect`.

NOTE There should be no unflushed data in the sink when `StreamConnect` is called.
If both the sink and source are of packet data streams, see Streaming packet-based data, then packet boundaries are preserved when data is transferred from the source to the sink. New boundaries may be inserted in the sink depending on the circumstances.
If only the sink is of a packet data stream type then arbitrary boundaries may be added.
A transform is automatically created when `StreamConnect` is called. It is not necessary to start or stop this transform by calling `TransformStart` or `TransformStop` from the application.

Connection example

The following call shows how a stream can be connected using the `PanicNull` function to check that the source and sink being connected are valid and successfully connected:

```
PanicNull(StreamConnect(StreamKalimbaSource(3),StreamHostSink(3)));
```

This function call attempts to connect the Host interface sink to Kalimba DSP port 3. If either the source or the sink is not valid, or if the connection fails, the application panics.

4.2 Transform creation functions

There are a number of functions that allow the creation of transforms that can connect sources to sinks. Which are available may depend on the product platform or firmware type. See the ADK `stream.h` API documentation for those available on your platform.

Table 4-1 Transform control functions

Transform Control Function	Description
`TransformStart()`	Newly created transforms must be started for data to flow from the source to the sink.
`TransformStop()`	Not used for transforms created from `StreamConnect()`, use `TransformDisconnect()` instead.
`TransformDisconnect()`	Disconnects and destroys the transform
`TransformPollTraffic()`	Reports if any traffic has been handled by a transform.
`TransformFromSource()`	Find and return any transform connected to the indicated source.
`TransformFromSink()`	Find and return any transform connected to the indicated sink.
`TransformConfigure()`	Used to directly configure transforms, configuration keys tend to be transform type specific for example, `VM_TRANSFORM_RPT_SBC_ENCODE_PACKET_SIZE.`

The `TransformConfigure()` function can be used to configure these transforms, once created. The `TranformFromSink()` or `TransformFromSource()` functions can be used to find the existing stream transform from the stream's sink or source identifier. See the `transform.h` API documentation for more details.

4.3 Transform control functions

Table 4-2 Transform control functions

`Transform Control Function`	Description
`TransformStart()`	Newly created transforms must be started for data to flow from the source to the sink.
`TransformStop()`	Not used for transforms created from StreamConnect(), use TransformDisconnect() instead.
`TransformDisconnect()`	Disconnects and destroys the transform
`TransformPollTraffic()`	Reports if any traffic has been handled by a transform.
`TransformFromSource()`	Find and return any transform connected to the indicated source.
`TransformFromSink()`	Find and return any transform connected to the indicated sink.
`TransformConfigure()`	Used to directly configure transforms, configuration keys tend to be transform type specific for example, `VM_TRANSFORM_RPT_SBC_ENCODE_PACKET_SIZE.`

See the `transform.h` API documentation for more details.

4.4 Direct connection stream compatibility

The table shows the types of source and sink that can be connected together with a direct connection to stream data between them. That is, those which may be used with `StreamConnect()` or otherwise linked by a transform.

Table 4-3

		Sink									
Source		Kalimba	Audio	SCO	RFCOMM	L2CAP	UART	Host	USB (1) (2)	Partition Write	Pipe
	Kalimba	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	-	Yes
	Audio	Yes	Yes	Yes	-	-	-	-	-	-	-
	SCO	Yes	Yes	-	-	-	-	-	-	-	-
	RFCOMM	Yes	-	-	Yes	Yes	Yes	Yes	Yes	-	Yes
	L2CAP	Yes	-	-	Yes	Yes	Yes	Yes	Yes	-	Yes
	UART	Yes	-	-	Yes	Yes	Yes	Yes	Yes	Yes	Yes
	Host	Yes	-	-	Yes	Yes	Yes	Yes	Yes	Yes	Yes
	USB (1) (2)	Yes	-	-	Yes	Yes	Yes	Yes	Yes	Yes	Yes
	Region	Yes	Yes	-	Yes	Yes	Yes	Yes	Yes	-	Yes
	File	Yes	Yes	-	Yes	Yes	Yes	Yes	Yes	-	Yes
	Ringtones	Yes	Yes	-	Yes	Yes	Yes	Yes	Yes	-	Yes
	I^2C	Yes	Yes	-	Yes	Yes	Yes	Yes	Yes	-	Yes
	Partition Raw Read	Yes	Yes	-	Yes	Yes	Yes	Yes	Yes	-	Yes
	Pipe	Yes	-	-	Yes	Yes	Yes	Yes	Yes	Yes	Yes

NOTE (1) There are three USB sink/source types; `UsbClass`, `UsbEndPoint` and `UsbVendor`, the entries apply to all three types.

(2) Isochronous USB endpoints work similarly to audio sinks and sources, that is, not like UART sources and sinks.

Where an application requires a connection between a sink and source that cannot be connected directly, the Kalimba DSP can be used to facilitate the connection, for example:

```
SCO source -> Kalimba sink, Kalimba source -> SCO sink
```

5 Managed connections between sinks and sources

In a managed connection, the application is responsible for processing data as well as controlling how it is moved from the source to the sink (the flow control).

The source functions are used to provide a stream of octects (type `uint8s` to the application. The sink functions provide a way for an application to send a stream of octects elsewhere. That is, when the application is receiving data, it is using the Source API to read data from a source. When the application produces data, it uses the Sink API to write data to a sink.

It is not practical for applications to poll a source for the presence of more data, or a sink for the ability to claim more space. Instead, the streams library sends messages to the application to indicate these states. .

NOTE Managed Source/Sink APIs cannot be used on a directly connected Source/Sink stream. Therefore, they return a failure on a directly connected source/sink streams.

For managed connections streams are buffered. That is, if there is data in a sink, it remains until removed by a flush. If there is data in a source, it remains until removed by a drop. If data is not flushed or dropped, the source or sink fills up until no more data can be written to it.

5.1 Streaming packet-based data

The creator of the data going into a source may make it a simple stream of data (for example, a file or a UART connection) or may divide the data into packets with boundaries between them, for example USB data. Each boundary may also have a small amount of data (for example, a packet header) attached to it.

NOTE A boundary and any associated header does not take up any space in the data stream, that is. it is out-of-band.
Therefore, functionality is provide in the API to:

- Add a packet header to any data written using the Sink API
- Look for the presence of boundaries and headers in any data read using the Source API
- Read headers for any data read using the Source API

6 What is a sink

A sink can be thought of as an infinite buffer divided into four sections.

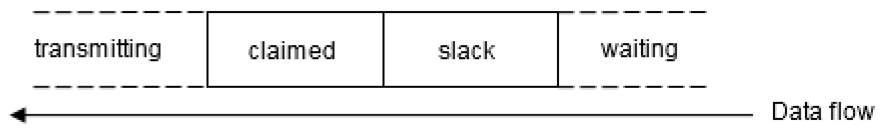

Figure 6-1 Sink buffer

- The transmitting section holds the data that the application has previously prepared and passed to the `stream` library that is, it has been flushed) and is now transmitting the data to its destination.
- The claimed section is the only area of the buffer that the application is permitted to access. It is available space to which the application may write. When a `sink` is initialized, no space is claimed.
- The slack section is space that the `stream` library is not using, and may be claimed by the application.
- The waiting section is space that, for whatever reason, the `stream` library is unable to make available yet.

Only one sink can be in the application address space at a time, and therefore the application can write data to one sink at a time. This is called the mapped sink. A sink is mapped using `SinkMap`.

In general, an application should only write to a sink upon receipt of a `MESSAGE_MORE_SPACE` message . This indicates that there is additional slack space in the sink, which can be claimed. An application writing to a sink performs the following sequence:

1. Obtain the amount of space available in the sink using `SinkSlack()`
2. Claim a portion of the slack using `SinkClaim()`
3. Map the claimed section into application memory using `SinkMap()`
4. Copy data into the mapped sink
5. Flush the sink using `SinkFlush()`

When the sink is flushed, data streams to its destination. At some point later, another `MESSAGE_MORE_SPACE` message is sent to the application and the process repeats.

NOTE There is a strict limit on how big the claimed section can grow (approximately 3 kB), so an application should only claim space when it is ready to use it. Only a limited amount can be mapped into the application address space at one time.
There is no way for an application to un-claim space when it has been claimed. It can only be flushed and the data in the claimed space transmitted. Applications therefore should not claim more space than needed.

See the sink functions declared in the `sink.h` and `system_message.h` header file.

7 What is a source

A source can be thought of as an infinite buffer divided into three sections.

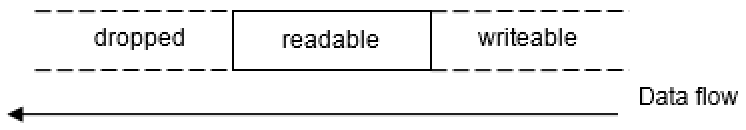

Figure 7-1 Source buffer

- The dropped section is data that the application has already processed and does not need the `stream` library to retain.
- The readable section holds the data that the `streams` library has placed in the `source` and that the application is permitted to access.
- The writeable section is where the library is still placing data.

In general, an application should only read from a source upon receipt of a `MESSAGE_MORE_DATA` message. This indicates that there is data in the source which can be read. An application reading from a source must perform the following sequence:

1. Get the amount of readable data in the source, using `SourceSize()`
2. Map the source into application memory, using `SourceMap()`
3. Process the data received
4. Drop processed data, using `SourceDrop()`

If more data arrives at the source, the process is repeated.

NOTE There is a strict limit on how big the readable section can grow, so applications should drop data as soon as possible. Failing to do so eventually chokes off the supply of new data. Even if the source is a data block in the firmware or the read-only filing system, only a limited amount (currently about 3 kB) can exist in the application address space at one time.

Packet-based streams

Packet boundaries and any associated headers, if present, are not part of the data stream.

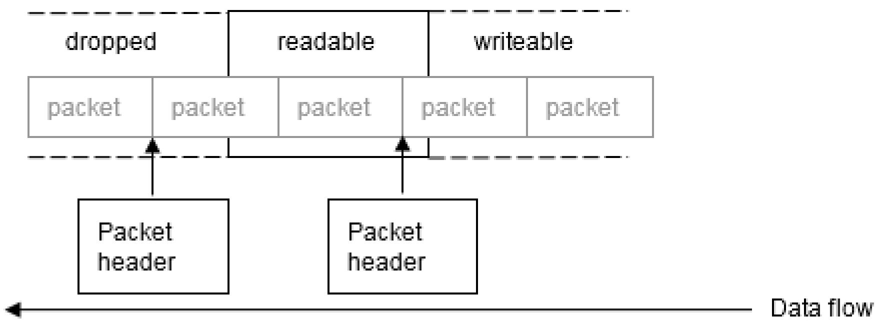

Figure 7-2 Packet-based stream

`SourceBoundary()` must be called to identify packet boundaries within a source, and `SourceSizeHeader()` and `SourceMapHeader()` must be called to map in any packet headers.

NOTE Not every boundary has a packet header associated with it.

See the `source.h` header file.

7.1 Example code to stream a simple string to a UART

This example code shows how a simple string can be streamed to a UART using a managed connection.

Figure 7-3 Streaming a string to a UART

```
const char *string = “Hello, World!”;
uint16 length = strlen(string);

/* Get the sink for the UART, panic if not available. */
Sink sink = StreamUartSink();
PanicNull(sink);

/* Claim space in the sink, getting the offset to it. */
uint16 offset = SinkClaim(sink, length);
if (offset == 0xFFFF) Panic();    /* Space is not available. */

/* Map the sink into memory space. */
uint8 *dest = SinkMap(sink);
(void) PanicNull(dest);

/* Copy the string into the claimed space. */
memcpy(dest+offset, string, length);

/* Flush the data out to the UART. */
PanicZero(SinkFlush(sink, length);
```

7.2 StreamMove() function

The `StreamMove()` function is declared in the `source.h` header file.

The `StreamMove()` function moves the specified number of bytes from the start of the specified source to the end of the specified `sink`. The number of octets to move must be no more than the size of the data in the source and the available space in the sink, as returned by `SourceBoundary()` and `SinkSlack()`.

This function combines the functionality of `SinkClaim()`, `memcpy()` and `sourceDrop()` into a single function. A call to `sinkFlush()` must still be made to actually transmit the data written to the `sink`.

`StreamMove` returns zero if the operation fails and the count if the operation succeeds.

NOTE A call to `StreamMove` does not invalidate any active `SinkMap` or `SourceMap`.

Figure 7-4 Example of the use of StreamMove()

```
/* Claim space in the sink. */
uint16 claim_result = SinkClaim(sink, number_of_octets);

/* If the claim failed return an error. */
If (claim_result == 0xFFFF) return 0;

/* Move data from source to sink. */
StreamMove(sink, Source, number_of_octets);

/* Flush the data. */
(void) SinkFlush(sink, number_of_octets);
```

7.3 Managed connection stream compatibility

Table 7-1 shows which types of stream are compatible with managed connections.

Table 7-1 Streams compatible with managed connections

Stream	Managed connection
Kalimba	No
Audio	No
SCO	No
RFCOMM	Yes
L2CAP	Yes
UART	Yes
Host	Yes
USB(1)	No

Table 7-1 Streams compatible with managed connections (cont.)

Stream	Managed connection
Region	Yes
File	Yes
Ringtones	Yes
I^2C	Yes
Partition	Yes
Pipe	Yes
(1) There are three USB sink/source types, `UsbClass`, `UsbEndPoint`, and `UsbVendor`, this entry applies to all three types.	

8 Streaming messages

When an application manages connections, the firmware uses various messages to indicate important stream related events. The application can also receive message notification in the event of a direct connection being disconnected so that the application can properly tidy up resources. See the `stream.h` and `system_message.h` API documentation for more details.

8.1 How to configure a task to receive stream related messages

To configure an application task to receive stream related messages, the source or sink identifier for the stream is required.

- `Task MessageStreamTaskFromSink(Sink sink, Task task)`: Use this function to configure the identified task to receive messages from the stream identified by the sink.
- `Task MessageStreamTaskFromSource(Source source, Task task)`: Use this function to configure the identified task to receive messages from the stream identified by the source.

> **NOTE** These functions return the task previously associated with the stream, if there was one, or `0`.

To find which application task (if any) is already associated with a source or sinks stream related messages, the following functions are available:

- `Task MessageStreamGetTaskFromSink(Sink sink)`
- `Task MessageStreamGetTaskFromSource(Sink source)`

8.2 Stream related messages

The following stream related messages are for managed connections:

- `MESSAGE_MORE_DATA`: Indicates that there is more data available to be read from the source of a stream. The message contains the source identifier.
- `MESSAGE_MORE_SPACE`: Indicates that there is more space available to write to in the sink of a stream. The message contains the sink identifier.
- `MESSAGE_SOURCE_EMPTY`: Indicates that the source associated with a stream has no more data to be read. The message contains the source identifier.

The following stream related message is for direct connections:

- `MESSAGE_STREAM_DISCONNECTED`: indicates the direct connection managed stream has disconnected. The message contains both the sink and source identifiers for the stream. If there is a task configured to receive messages identified with the sink and a separate task configured to receive messages identified with the source, then both tasks will receive this message, that is, this message is sent twice, once to the source related task and once to the sink related task.

9 Common source and sink functionality

Functions are available for configuring and monitoring sources and sinks in both managed and direct connections.

Three functions are available for configuring sources, sinks and streams:

- `SinkConfigure()`: Declared in the `sink.h` header file
- `SourceConfigure()`: Declared in the `source.h` header file
- `StreamConfigure()`: Declared in the `stream.h` header file

Three functions are used to link together two sources or two sinks. They are typically useful for managing audio streams:

- `bool SourceSynchronise(Source source1, Source source2)`: Synchronize two source streams. This makes it possible to keep two sources associated with the stereo A channel and the stereo B channel synchronized.
- `bool SinkSynronise(Sink sink1, Sink sink2)`: Synchronize to sink streams.
- `bool SinkAlias (Sink sink1, Sink sink2)`: Alias sink1 to sink2 so that sink2 receives the same audio data from the same source as sink1.

Refer to the library API documentation for more detailed information on each function.

Example code

The example code shows how sources and sinks may be linked. The example shows stereo audio loopback using the codec hardware.

Figure 9-1 Example of code linking sources and sinks

```
/* A simple loopback of codec to codec. */

Source srcA;
Source srcB;
Sink sinkA;
Sink sinkB;

/* Get the sinks and sources, related to audio HW. */
srcA = StreamAudioSource(
            AUDIO_HARDWARE_CODEC,
            AUDIO_INSTANCE_0,
            AUDIO_CHANNEL_A);
```

```
srcB = StreamAudioSource(
            AUDIO_HARDWARE_CODEC,
            AUDIO_INSTANCE_0,
            AUDIO_CHANNEL_B);

sinkA = StreamAudioSink(
            AUDIO_HARDWARE_CODEC,
            AUDIO_INSTANCE_0,
            AUDIO_CHANNEL_A);
sinkB = StreamAudioSink(
            AUDIO_HARDWARE_CODEC,
            AUDIO_INSTANCE_0,
            AUDIO_CHANNEL_B);

/* Sychronize the A and B stereo channels. */
SourceSynchronise(srcA, srcB);
SinkSynchronise(sinkA, sinkB);

/* Create the codec connection.*/
SourceConnect(srcA, sinkA);
```

9.1 Functions that return information about a stream

The following functions can be used to return useful information on a stream:

- `Sink StreamSinkFromSource(Source source)`: Return the sink for the same stream as the source. This function is useful as most of the other stream information functions use the sink as the identifier.
- `uint16 SinkGetScoHandle(Sink sink)`: Return the SCO handle for the connection.
- `uint16 SinkGetRfcommConnId(Sink sink)`: Return the `conn_id` of the RFCOMM connection.
- `uint16 SinkGetL2capCid(Sink sink)`: Return the cid of the L2CAP connection.
- `bool SinkGetBdAddr(Sink sink, typed_bdaddr *taddr)`: Get the Bluetooth device address, type and transport type, for the stream identified by the sink, in the taddr argument, which is passed by reference.
- `bool SinkGetRssi(Sink sink, int16 *rssi)`: Get the receive signal strength indication for the stream identified by the sink, in the `rssi` argument, which is passed by reference.
- `bool SinkPollAwayTime(Sink sink, uint16 *msec)`: Get the Time Away of an ACL identified by the sink, in the msec argument, which is passed by reference. The Time Away is the time since any packet was last received on that ACL.

NOTE If the input identifier or stream type is not the correct, these functions return `0` or `FALSE`.

The following functions can be used to check if the source or sink identifier is valid.

- `bool SinkIsValid(Sink sink)`
- `bool SourceIsValid(Source source)`

10 Technical support

Further information on all QTIL products can be found on the createpoint website.

Document references

Document	Reference
On-line VM and Native library reference documentation in xIDE	n/a

Terms and Definitions

Term	Definition
AAC	Advanced Audio Coding
ACL	Asynchronous ConnectionLess
ADK	Audio or Application Development Kit
ADPCM	Adaptive Differential Pulse code Modulation
API	Application Programming Interface
ATRAC	Adaptive Transform Acoustic Coding
BCSP	BlueCore Serial Protocol
Bluetooth SIG	Bluetooth Special Interest Group
Bluetooth	Set of technologies providing audio and data transfer over short-range radio connections
CODEC	COder DECoder
DAC	Digital to Analog Converter
DSP	Digital Signal Processor
I^2C	Inter-Integrated Circuit
I^2S	Inter-Integrated Circuit Sound
IC	Integrated Circuit
ID	Identifier
L2CAP	Logical Link Control and Adaptation Protocol
MP3	MPEG-1 audio layer 3
MPEG	Moving Picture Experts Group
PCM	Pulse Code Modulation
Pipe	A bi-directional stream for data transfer from one end to another
QTIL	Qualcomm Technologies International, Ltd.
RFCOMM	Serial cable emulation protocol
RSSI	Received Signal Strength Indication
RTP	Real Time Protocol
SBC	Sub Band Coding
SCO	Synchronous Connection-Oriented link
SDK	Software Development Kit

Term	Definition
SPDIF	Sony/Philips Digital InterFace (also IEC 958 type II, part of IEC-60958). An interface designed to transfer stereo digital audio signals between various devices and stereo components with minimal loss.
UART	Universal Asynchronous Receiver Transmitter
USB	Universal Serial Bus
VM	Virtual Machine
xIDE	The QTIL Integrated Development Environment