Architecture Overview

Application Programming Interface Reference Manual

Release: 4.0.1
January 10, 2013

Louisville, KY www.stonestreetone.com

Bluetopia Architecture Overview

1. Introduction

1.1 Purpose

The purpose of this document is to identify and explain the overall architecture of Bluetopia, the Bluetooth Protocol Stack by Stonestreet One. This document will provide a small overview of Bluetopia followed by a more thorough architecture overview based upon the current implementation. This document does not serve as a Bluetooth Primer and does not attempt to explain the various Bluetooth Protocols/Profiles mentioned in this document.

2. General Description

2.1 Overview

Bluetopia is Stonestreet One's upper layer Bluetooth Protocol Stack implementation. The core stack consists of the following layers/protocols/profiles:

- HCI (protocol)
- L2CAP (protocol)
- SDP (protocol)
- RFCOMM (protocol)
- SPP (Serial Port Profile)
- GAP (Generic Access Profile)
- OBEX (protocol)

The architecture depicted in Figure 1, depicts a modular structure such that any module can be easily removed without impacting any other modules. Obviously if a protocol layer is removed, then protocols/profiles that exist above that protocol that use the removed underlying protocol would have to be removed as well. For example, if RFCOMM is removed, then SPP would have to be removed because SPP uses RFCOMM as the underlying protocol. The other modules (BSC, SDP, SCO, L2CAP, GAP, and HCI), however, will not be impacted by the removal of this layer. The reader should note that every layer that uses a lower layer protocol/profile uses the same API that is published in the *Bluetopia API Reference Manual* by Stonestreet One. This further illustrates the ease with which modules can be added/removed/replaced.

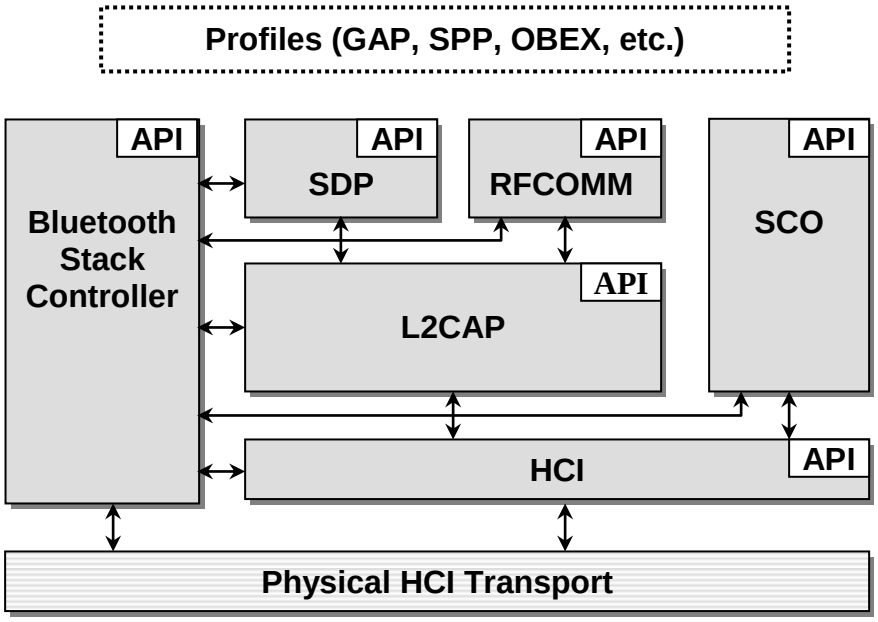

Figure 1 – Bluetopia architecture.

3. Architecture Overview

3.1 Design Overview

Bluetopia is designed such that the application programmer does not have to be aware of implementation details. To satisfy this requirement, Bluetopia is designed based on the following principles:

- Fully thread safe (threaded environments)
- Asynchronous event notification
- Fully Re-entrant
- Entirely self contained (threaded environments)

The first item, being thread safe, means that multi-threaded applications can make Bluetopia calls without the need for synchronization events to guard access to the Bluetooth Protocol Stack (unless the application is designed such that it requires its own synchronization). The second item, aids multi-threaded applications in that making a Bluetopia API call does not block the calling thread for long periods of time and requires no polling. Bluetopia is designed such that the application is notified of asynchronous events via application installed callbacks. These callbacks are called at the time an event occurs. The third item, re-entrancy, means that all Bluetopia API calls are safe to be called from the asynchronous callback functions. This simplifies application development by not requiring the programmer to define messages or events based upon stack events (and having to notify the application of these events).

The final item, entirely self contained, means that the programmer does not have to call any functions to cause Bluetopia to function. An example of this would be a function to 'pump' a state machine. Bluetopia requires no such actions when operating in threaded environments. When Bluetopia is operating in a single threaded environment, functions will need to be called to periodically to process the Bluetooth state machine.

3.2 Asynchronous Event Overview

The Bluetooth Protocol Stack, like all networking protocol stacks, is very asynchronous in nature. This implies that some asynchronous notification schema is implemented to notify the host application of events that may have happened. Bluetopia implements its Event Notification schema via callback functions. In general, a callback function has the following form:

void BTPSAPI **Event_Callback**(Event_Data_t * Event_Data, unsigned long CallbackParameter);

Where the **Event_Data** parameter is specific to the type of callback that is registered and the **CallbackParameter** parameter is a user defined value that is passed when the callback is installed. Each Protocol/Profile level in the protocol stack defines the required callbacks and required events for the Protocol/Profile. As previously stated, any Bluetopia function can be called within this function.

An example of an asynchronous event would be an L2CAP Connection or HCI Event or ACL Data. The programmer could register for a specific event using the corresponding API and the application would be notified when this event occurred.

3.3 Thread Overview

It should be obvious that for Asynchronous events to be dispatched, there must be a dispatcher that executes. Bluetopia requires at least one thread to function (not including application and/or transport threads). This thread exists to dispatch asynchronous events. In general, Bluetopia implements a second thread, an asynchronous timer thread to handle the timing out of Bluetooth Actions/Events. This timer thread can be removed if the Operating System provided allows some sort of asynchronous timing (i.e. a mechanism to install a timer into the system and be notified when the timer has expired). This thread can be collapsed into the dispatch thread to yield only one thread required for Bluetopia.

The dispatch thread mentioned above, executes when there is data to be processed from the Bluetooth Device. How this data arrives to the stack can be from any number of ways, including:

- Physical Transport (UART/USB/Synchronous Serial, etc.)
- Direct callback from Base-band controller

When data arrives it is sent to the dispatch thread. This serves the purpose of allowing the transport driver to continue to function and allows further data to be processed, allowing re-entrant stack calls to be made from dispatched event callbacks. This thread ONLY exists for this purpose, that is, to not hang the transport thread.

The dispatch thread then processes the data, running in the HCI Layer of the Bluetooth Protocol Stack. The HCI layer then dispatches any necessary events to higher layer protocols/profiles that might be registered via registered callbacks.

As mentioned previously, a second thread may be present. This thread is responsible for handling various Bluetooth timing functions. In practice, this thread is relegated to the Operating System Abstraction layer so the application programmer has access to a generic asynchronous timing interface.

When operating in a non-threaded environment, the above architecture is still valid. The only difference is that the dispatcher thread isn't really a thread at all (nor is the timer thread), rather events are dispatched directly out of the function that is periodically called to process the Bluetopia/Bluetooth state machine.

3.4 Operating System Abstraction Overview

Bluetopia uses a generic Operating System Abstraction layer to aid in both porting Bluetopia and applications using Bluetopia.

Further information regarding the functionality of this layer can be found in the *Bluetopia System Call Requirements* documents by Stonestreet One. Note that there are two different versions of the above listed document. One for multi-threaded environments and another for single threaded environments.

3.5 Functional Overview

As stated earlier, all levels of the Bluetooth Protocol Stack, depicted in Figure 1, are self contained. Each level encapsulates all processing required for the functionality contained in that level. For example, L2CAP handles all segmentation/re-assembly tasks, and the lower layer (HCI) and upper layers (SDP, RFCOMM, etc.) are not concerned with this internal functionality.

This encapsulation means that each layer might require RAM to buffer/queue data. Currently, each individual layer is responsible for its own memory allocation/deallocation. No layer allocates memory that is deallocated by another (higher or lower) layer. This approach leads to more potential memory copying, however allows encapsulation at each layer and simplifies debugging. It is possible to optimize the current schema to use a common packet/data pool if required. In practice, the current relatively low bandwidth of Bluetooth has not shown this design decision to be a problem.