System Call Requirements

(Multi-threaded O/S)

Application Programming Interface Reference Manual

Release: 2.1.3
January 19, 2011

Louisville, KY www.stonestreetone.com

TABLE OF CONTENTS

Bluetopia System Call Requirements

1. Introduction

1.1 Purpose

The purpose of this document is to identify and explain the necessary requirements that will be needed to effectively run Bluetopia, the Bluetooth Protocol Stack by Stonestreet One, on a specific Processor/Operating System. This document will provide a small overview of Bluetopia and the Operating Systems it has been successfully ported to, followed by a list of system functions that are called based upon the current implementation.

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.

As of this writing, all above layers of Bluetopia have been ported (and successfully tested) on the following platforms:

- Windows (Intel x86, Windows 95/98/ME/NT/2000/XP)
- Windows CE (Strong ARM, Hitachi SH3/SH4)
- Linux (Intel x86)
- QNX (Intel x86)
- VxWorks (Hitachi SH4)
- DSP BIOS (Texas Instruments C5416 DSP)
- RTKernel (Intel x86, MS-DOS)
- Generic/proprietary Operating System:
  - Mitsubishi M16C
  - Texas Instruments C5416 DSP
  - ARM7/ARM9/ARM11

It should be noted that all of the above listed Bluetopia ports share the same API among all platforms. Bluetopia has been designed for ease of use, while maintaining easy portability. Bluetopia has been written in highly portable ANSI-C. In all of the above cases, only the Physical HCI Transport Layer and the Bluetooth/Kernel Interface module were rewritten for each port. It should be noted that the Bluetooth/Kernel Interface is NOT shown in Figure 1.

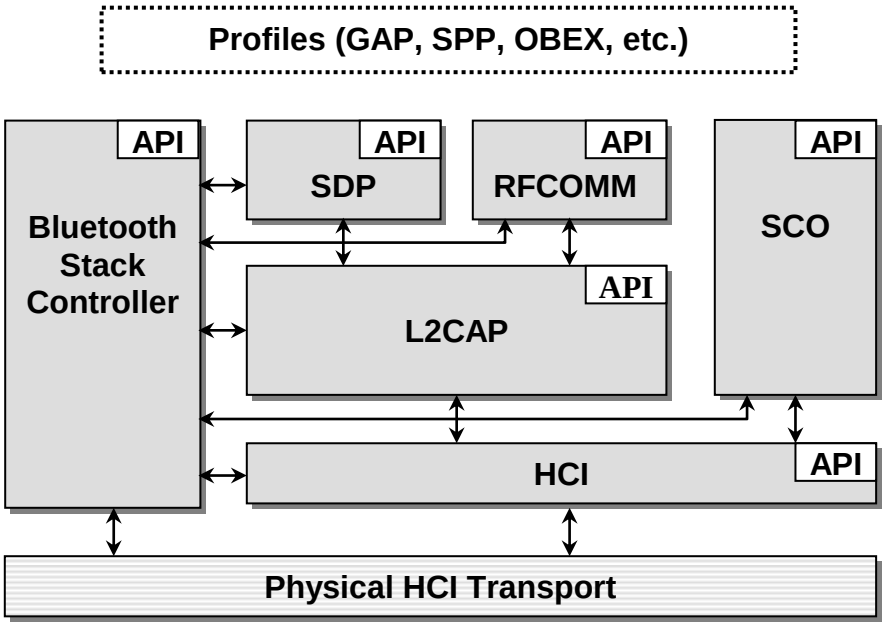

Figure 1 – Bluetopia architecture.

In addition to the standard Bluetooth Core Protocol Stack, Bluetopia can support many of the upper layer Bluetooth profiles. Profiles which can be supported include:

Advance Audio Distribution Profile (A2DP)

Audio/Video Control Transport Protocol (AVCTP)
Audio/Video Distribution Transport Protocol (AVDTP, implemented as GAVD)
Audio/Video Remote Control Profile (AVRCP)
Basic Imaging Profile (BIP)
Basic Printing Profile (BPP)
Dial-up Network Profile (DUN)
File Transfer Profile (FTP)
Hardcopy Cable Replacement Profile (HCR)
Headset Profile (HDS)
Hands Free Profile (HFRE)
Human Interface Device Profile (HID)
Object Exchange Profile (OBP)
Message Access Profile (MAP)
Personal Area Network Profile (PAN)
Phonebook Access Profile (PBAP)
SIM Access Profile (SAP)
Synchronization Profile (SYNC)

Several of these profiles require platform support of file and folder (or directory) storage concepts. In order to support these capabilities, an additional platform file abstraction layer has been defined in Bluetopia. Since profiles may be included, but are not necessarily included in any given Bluetooth product implementation, implementation of this module is required only when a product includes a profile which needs these underlying platform services.

2.2 Host Controller Interface Vendor Specific Extensions

Bluetopia is designed to be HCI neutral, meaning it functions with any HCI hardware implementation. However, Stonestreet One's upper layer Bluetooth Protocol Stack is designed to facilitate implementing extensions that take advantage of HCI vendor specific capabilities. This is accommodated via the **HCI_Send_Raw_Command** documented in the BSPAPI documentation, as well as in functions called from Bluetopia during initialization to allow implementation of vendor specific functionality. These extension functions are defined in the following sections of this document.

3. Interface Requirements

3.1 Software Interfaces

As previously mentioned, porting Bluetopia to a new platform/architecture can be accomplished by revising only two modules. These modules are the physical HCI Transport Layer and the Bluetooth/Kernel Interface module. A third module may need to be provided, assuming a profile is incorporated which requires underlying file and folder (directory) storage support. Because the HCI Transport Layer is so

heavily Operating System and platform specific this document will not go into great detail about the workings of this module. In the current versions of Bluetopia, this module simply sends entire Bluetooth HCI Command/ACL/SCO data packets and builds HCI Event/ACL/SCO data packets which are then handed to the HCI Layer of the stack. The HCI Transport layer is a very small layer with a very small, defined, API that can be adapted to different transport mechanisms.

The Bluetooth/Kernel Interface module is designed to abstract Operating System dependent service primitives from the Bluetooth Protocol Stack implementation. These primitives can be lumped into the following categories:

- Event Handling Functions
- Mutex Handling Functions
- Timer Handling Functions
- Thread Handling Functions
- Memory Handling Functions
- String Handling Functions
- Miscellaneous Functions

Each of the above categories will be explained further in its own section below. Each section will list the prototypes of the Kernel Interface functions. The actual Source Code Header File can be found in the Section 4. It should be noted, that if the actual operating system does NOT support some of the specified functionality, other Operating System primitives will be used to achieve the desired functionality. For example, many Operating Systems do not provide a Mutex primitive, and the Bluetooth/Kernel Interface primitive is coded using Operating System Semaphore primitives to achieve the desired functionality.

3.1.1 Event Handling Functions

The definition of an Event is similar to a Binary Semaphore in that the Event only has two states. An event can either be “Set” (signaled) or “Reset” (un-signaled). The Event primitive also allows the programmer the option of waiting (blocking) for an Event to become “Set” (signaled). The programmer can issue the following functions relating to Events:

- Create/Destroy an Event
- Set/Reset the State of Event
- Wait for an Event to be Signaled

3.1.2 Mutex Handling Functions

The definition of a Mutex is similar to a normal Semaphore, except that if the calling thread already owns the Mutex and attempts to acquire the Mutex again it

will be granted immediately (no deadlock will occur). The Mutex is similar to the Semaphore in that for every call to Acquire there must be a corresponding Release call or a deadlock can occur. The programmer can issue the following functions related to Mutexes:

- Create/Destroy a Mutex
- Acquire/Release a Mutex

3.1.3 Timer Handling Functions

Bluetopia implements its own timing mechanism in a Timer Module. This Timer Module uses the primitives of the Bluetooth/Kernel Interface Module. The only timing primitive required for the Bluetooth/Kernel Interface Module is a function to delay a specified number of milliseconds. The programmer can issue the following functions related to Timing:

- Wait (Block) for the specified number of milliseconds.

It should be noted that the Timer Module has its own API and only relies on the above listed primitive to be provided by the Operating System.

3.1.4 Thread Handling Functions

Bluetopia spawns two threads when it executes. These threads use the Create Thread primitive that is part of the Bluetooth/Kernel Interface. A Thread cannot be killed once it is started (unless it is signaled to do so by some mechanism, in which case it will simply run to completion (return)). The programmer can issue the following functions related to Threads:

- Create a new Thread
- Query the currently executing Thread Handle/ID.

Bluetopia creates a thread used for dispatching asynchronous timers and another thread for dispatching HCI Events/ACL/SCO packets throughout the stack. If the underlying Operating System provides timers, then it is possible to remove the Bluetooth Timer thread completely.

3.1.5 Memory Handling Functions

The memory handling requirements for Bluetopia are similar to any memory handling support that would be found with any Operating System and/or run-time library. The programmer can issue the following memory handling functions:

- Allocate/Free Memory
- Copy a Memory Buffer
- Initialize/Fill a Memory Buffer with a value
- Compare two Memory Buffers
- Query Memory Usage

Bluetopia declares and manages a fixed pool of memory buffers. The sizes of the buffers are fixed using a data structure in the Bluetooth/Kernel Interface module. The initialization and management of these memory buffer pools are transparent to the programmer. For optimization purposes, the Query Memory Usage API is provided to allow the programmer to determine current and maximum allocations from the various buffer sizes. In order to optimize memory, the programmer could exercise the foreseen stack functions then issue this call to determine if more or less buffers of a particular size could be configured to optimize memory allocation.

3.1.6 String Handling Functions

Bluetopia requires a minimal set of string handling functions. The programmer can issue the following string handling functions:

- Copy a NULL terminated ASCII String
- Determine the length of a NULL terminated ASCII String
- sprintf() type function (only required if OBEX Layers are used)

3.1.7 Miscellaneous Functions

The category called miscellaneous functions are primitives that are provided to Bluetopia that are built upon the above listed primitives. These primitives are:

- Mailbox Primitives
- Memory Pool Primitives

These primitives are currently built using Bluetooth/Kernel Interface Primitives (discussed previously) and are only mentioned because the reader will notice these prototypes in the source header file listed in the appendix.

3.1.8 Optional Support Functions

The category called optional support functions are primitives that are provided to Bluetopia that are used only for support purposes during porting or other development activities. These functions must be defined to insure an operation system is constructed. However, the functionality is not strictly required unless

Stonestreet One support personnel need to obtain additional information at run-time during the stack operation. These primitives are:

- Initialization functions
- Diagnostic message display function

3.1.9 Vendor Specific HCI Support Functions

The category of vendor Specific HCI Support functions are a set of six functions called during stack initialization. A default implementation of these functions is provided in the HCI layer of Bluetopia. This default implementation essentially performs no operations. However, a specific version of these functions can be written and included in a system prior to linking the Bluetopia stack to allow taking advantage of vendor specific extensions, or to perform additional vendor specific HCI initialization sequences to achieve optimum operation with a specific HCI device. These extensions and possible uses scenarios are defined in the following table. The BTPSVEND.h header file contains the API definition of these calls.

Function call point	Possible use scenario
HCI_VS_InitializeBeforeHCIOpen	Enabling HCI interface hardware circuitry, powering up the HCI device.
HCI_VS_InitializeAfterHCIOpen	Peforming any post HCI interface initialization unique to the platform which doesn't require HCI communications via the stack, powering up the HCI part.
HCI_VS_InitializeBeforeHCIReset	Performing any HCI platform initialization sequences that require the use of the stack HCI communication layer. Note that the HCI device WILL BE RESET via the HCI_Reset function after this call, so any updates stored in HCI RAM may be lost.
HCI_VS_InitializeAfterHCIReset	Performing any HCI platform initialization sequences that require the use of the stack HCI communication layer. Note that the HCI_Reset function will have been completed prior to this call, so any updates stored in HCI RAM should not be lost.
HCI_VS_InitializeBeforeHCIClose	Performing any platform specific operations, perhaps updating the user interface or disabling access to Bluetooth functionality.
HCI_VS_InitializeAfterHCIClose	Disabling HCI interface circuitry, powering off the HCI, etc.

3.2 Porting Guidelines

Bluetopia is designed for easy migration to new platforms. This section provides a general, high level guideline of the steps required to successfully migrate Bluetopia to a new platform.

- Modify the Bluetopia/Kernel Interface module (BTPSKRNL.c) file to use target platform primitive operations to implement the functions described above. The miscellaneous primitives do not require system dependent coding, as they are built on top of the other primitives. Other primitives may require modification to match with the target system capabilities.
- Create a physical HCI transport layer implementation that uses the target platform primitive operations and hardware. Reference the Stonestreet One provided HCITRANS document for additional details on this component.
- Modify the Bluetopia/Vendor specific Interface module (BTPSVEND.c) file to support any vendor specific operations for the target Bluetooth chipset. This may entail simply stubbing the functions with no code for some implementations.
- Compile these modules, along with the Bluetopia stack modules for a complete Bluetooth core stack layer.
- Compile and use the stack demonstration programs on the target platform to verify proper operation.
    - Begin by exercising the stack reset/startup functionality. Use debugging techniques appropriate to the platform to verify that the initial reset sequence sent to the HCI controller is sent and received properly by the physical HCI transport layer.
    - Continue by exercising lower level HCI functions as incorporated in the HCI demonstration program.
    - When HCI level functionality has been validated, continue by exercising GAP and SDP functionality.
    - Verify SPP functionality. Note that this incorporates validation of the RFCOMM layer functionality.
    - Continue this approach with additional stack layers as relevant to the target platform.
- When all functional operation has been validated, perform any required performance verification and optimization by transferring larger amounts of data, starting appropriate protocol layer services, and validating memory usage statistics. Make any desired adjustments in the memory pool buffer declarations (if using static memory pools to support memory allocation), then compile and verify the final stack in operation.

4. Bluetooth/Kernel Interface Header File

```
/*****< bkrnlapi.h >*************************************************************/
/*      Copyright 2000 - 2010 Stonestreet One.                                   */
/*      All Rights Reserved.                                                     */
```

```
/*                                                                           */
/*  BKRNLAPI - Stonestreet One Bluetooth Stack Kernel API Type Definitions,  */
/*             Constants, and Prototypes.                                    */
/*                                                                           */
/*  Author:  Damon Lange                                                     */
/*                                                                           */
/*** MODIFICATION HISTORY ****************************************************/
/*                                                                           */
/*   mm/dd/yy  F. Lastname    Description of Modification                    */
/*   --------  -----------    ------------------------------------------------*/
/*   05/30/01  D. Lange       Initial creation.                              */
/*****************************************************************************/
#ifndef __BKRNLAPIH__
#define __BKRNLAPIH__

#include <stdio.h>              /* sprintf() prototype.                      */

#include "BTAPITyp.h"           /* Bluetooth API Type Definitions.           */
#include "BTTypes.h"            /* Bluetooth basic type definitions          */

   /* Miscellaneous Type definitions that should already be defined,    */
   /* but are necessary.                                                */
#ifndef NULL
   #define NULL ((void *)0)
#endif

#ifndef TRUE
   #define TRUE (1 == 1)
#endif

#ifndef FALSE
   #define FALSE (0 == 1)
#endif

   /* The following preprocessor definitions control the inclusion of   */
   /* debugging output.                                                 */
   /*                                                                   */
   /*    - DEBUG_ENABLED                                                */
   /*         - When defined enables debugging, if no other debugging   */
   /*           preprocessor definitions are defined then the debugging */
   /*           output is logged to a file (and included in the         */
   /*           driver).                                                */
   /*                                                                   */
   /*          - DEBUG_ZONES                                            */
   /*              - When defined (only when DEBUG_ENABLED is defined)  */
   /*                forces the value of this definition (unsigned long)*/
   /*                to be the Debug Zones that are enabled.            */
#define DBG_ZONE_CRITICAL_ERROR          (1 << 0)
#define DBG_ZONE_ENTER_EXIT              (1 << 1)
#define DBG_ZONE_BTPSKRNL                (1 << 2)
#define DBG_ZONE_GENERAL                 (1 << 3)
#define DBG_ZONE_DEVELOPMENT             (1 << 4)
#define DBG_ZONE_SHA                     (1 << 5)
#define DBG_ZONE_BCSP                    (1 << 6)
#define DBG_ZONE_VENDOR                  (1 << 7)

#define DBG_ZONE_ANY                     ((unsigned long)-1)

#ifndef DEBUG_ZONES
   #define DEBUG_ZONES                   DBG_ZONE_CRITICAL_ERROR
#endif

#ifndef MAX_DBG_DUMP_BYTES
   #define MAX_DBG_DUMP_BYTES            (((unsigned int)-1) - 1)
#endif

#define DEBUG_ENABLED

#ifdef DEBUG_ENABLED
   #define DBG_MSG(_zone_, _x_)          do { if(BTPS_TestDebugZone(_zone_))
BTPS_OutputMessage _x_; } while(0)
   #define DBG_DUMP(_zone_, _x_)         do { if(BTPS_TestDebugZone(_zone_))
BTPS_DumpData _x_; } while(0)
#else
   #define DBG_MSG(_zone_, _x_)
   #define DBG_DUMP(_zone_, _x_)
```

```
#endif

   /* The following constant defines a special length of time that      */
   /* specifies that there is to be NO Timeout waiting for some Event   */
   /* to occur (Mutexes, Semaphores, Events, etc).                      */
#define BTPS_INFINITE_WAIT                              (0xFFFFFFFF)

#define BTPS_NO_WAIT                                     0

   /* The following type definition defines a BTPS Kernel API Event     */
   /* Handle.                                                           */
typedef void *Event_t;

   /* The following type definition defines a BTPS Kernel API Mutex     */
   /* Handle.                                                           */
typedef void *Mutex_t;

   /* The following type definition defines a BTPS Kernel API Thread    */
   /* Handle.                                                           */
typedef void *ThreadHandle_t;

   /* The following type definition defines a BTPS Kernel API Mailbox   */
   /* Handle.                                                           */
typedef void *Mailbox_t;

   /* The following MACRO is a utility MACRO that exists to calculate   */
   /* the offset position of a particular structure member from the     */
   /* start of the structure.  This MACRO accepts as the first          */
   /* parameter, the physical name of the structure (the type name, NOT */
   /* the variable name).  The second parameter to this MACRO represents*/
   /* the actual structure member that the offset is to be determined.  */
   /* This MACRO returns an unsigned integer that represents the offset */
   /* (in bytes) of the structure member.                               */
#define BTPS_STRUCTURE_OFFSET(_x, _y)              ((unsigned int )&(((_x *)0)->_y))

   /* The following type declaration represents the Prototype for a     */
   /* Thread Function.  This function represents the Thread that will   */
   /* be executed when passed to the BTPS_CreateThread() function.      */
   /* * NOTE * Once a Thread is created there is NO way to kill it.  The*/
   /*          Thread must exit by itself.                              */
typedef void *(BTPSAPI *Thread_t)(void *ThreadParameter);

   /* The following function is responsible for delaying the current    */
   /* task for the specified duration (specified in Milliseconds).      */
   /* * NOTE * Very small timeouts might be smaller in granularity than */
   /*          the system can support !!!!                              */
BTPSAPI_DECLARATION void BTPSAPI BTPS_Delay(unsigned long MilliSeconds);

#ifdef INCLUDE_BLUETOOTH_API_PROTOTYPES
   typedef void (BTPSAPI *PFN_BTPS_Delay_t)(unsigned long MilliSeconds);
#endif

   /* The following function is responsible for creating an actual      */
   /* Mutex (Binary Semaphore).  The Mutex is unique in that if a       */
   /* Thread already owns the Mutex, and it requests the Mutex again    */
   /* it will be granted the Mutex.  This is in Stark contrast to a     */
   /* Semaphore that will block waiting for the second acquisition of   */
   /* the Sempahore.  This function accepts as input whether or not     */
   /* the Mutex is initially Signalled or not.  If this input parameter */
   /* is TRUE then the caller owns the Mutex and any other threads      */
   /* waiting on the Mutex will block.  This function returns a NON-NULL*/
   /* Mutex Handle if the Mutex was successfully created, or a NULL     */
   /* Mutex Handle if the Mutex was NOT created.  If a Mutex is         */
   /* successfully created, it can only be destroyed by calling the     */
   /* BTPS_CloseMutex() function (and passing the returned Mutex        */
   /* Handle).                                                          */
BTPSAPI_DECLARATION Mutex_t BTPSAPI BTPS_CreateMutex(Boolean_t CreateOwned);

#ifdef INCLUDE_BLUETOOTH_API_PROTOTYPES
   typedef Mutex_t (BTPSAPI *PFN_BTPS_CreateMutex_t)(Boolean_t CreateOwned);
#endif

   /* The following function is responsible for waiting for the         */
   /* specified Mutex to become free.  This function accepts as input   */
   /* the Mutex Handle to wait for, and the Timeout (specified in       */
   /* Milliseconds) to wait for the Mutex to become available.  This    */
```

```
   /* function returns TRUE if the Mutex was successfully acquired and  */
   /* FALSE if either there was an error OR the Mutex was not           */
   /* acquired in the specified Timeout.  It should be noted that       */
   /* Mutexes have the special property that if the calling Thread      */
   /* already owns the Mutex and it requests access to the Mutex again  */
   /* (by calling this function and specifying the same Mutex Handle)   */
   /* then it will automatically be granted the Mutex.  Once a Mutex    */
   /* has been granted successfully (this function returns TRUE), then  */
   /* the caller MUST call the BTPS_ReleaseMutex() function.            */
   /* * NOTE * There must exist a corresponding BTPS_ReleaseMutex()     */
   /*          function call for EVERY successful BTPS_WaitMutex()      */
   /*          function call or a deadlock will occur in the system !!! */
BTPSAPI_DECLARATION Boolean_t BTPSAPI BTPS_WaitMutex(Mutex_t Mutex, unsigned long
Timeout);

#ifdef INCLUDE_BLUETOOTH_API_PROTOTYPES
   typedef Boolean_t (BTPSAPI *PFN_BTPS_WaitMutex_t)(Mutex_t Mutex, unsigned long
Timeout);
#endif

   /* The following function is responsible for releasing a Mutex that  */
   /* was successfully acquired with the BTPS_WaitMutex() function.     */
   /* This function accepts as input the Mutex that is currently        */
   /* owned.                                                            */
   /* * NOTE * There must exist a corresponding BTPS_ReleaseMutex()     */
   /*          function call for EVERY successful BTPS_WaitMutex()      */
   /*          function call or a deadlock will occur in the system !!! */
BTPSAPI_DECLARATION void BTPSAPI BTPS_ReleaseMutex(Mutex_t Mutex);

#ifdef INCLUDE_BLUETOOTH_API_PROTOTYPES
   typedef void (BTPSAPI *PFN_BTPS_ReleaseMutex_t)(Mutex_t Mutex);
#endif

   /* The following function is responsible for destroying a Mutex that */
   /* was created successfully via a successful call to the             */
   /* BTPS_CreateMutex() function.  This function accepts as input the  */
   /* Mutex Handle of the Mutex to destroy.  Once this function is      */
   /* completed the Mutex Handle is NO longer valid and CANNOT be       */
   /* used.  Calling this function will cause all outstanding           */
   /* BTPS_WaitMutex() functions to fail with an error.                 */
BTPSAPI_DECLARATION void BTPSAPI BTPS_CloseMutex(Mutex_t Mutex);

#ifdef INCLUDE_BLUETOOTH_API_PROTOTYPES
   typedef void (BTPSAPI *PFN_BTPS_CloseMutex_t)(Mutex_t Mutex);
#endif

   /* The following function is responsible for creating an actual      */
   /* Event.  The Event is unique in that it only has two states.  These*/
   /* states are Signalled and Non-Signalled.  Functions are provided   */
   /* to allow the setting of the current state and to allow the        */
   /* option of waiting for an Event to become Signalled.  This function*/
   /* accepts as input whether or not the Event is initially Signalled  */
   /* or not.  If this input parameter is TRUE then the state of the    */
   /* Event is Signalled and any BTPS_WaitEvent() function calls will   */
   /* immediately return.  This function returns a NON-NULL Event       */
   /* Handle if the Event was successfully created, or a NULL Event     */
   /* Handle if the Event was NOT created.  If an Event is successfully */
   /* created, it can only be destroyed by calling the BTPS_CloseEvent()*/
   /* function (and passing the returned Event Handle).                 */
BTPSAPI_DECLARATION Event_t BTPSAPI BTPS_CreateEvent(Boolean_t CreateSignalled);

#ifdef INCLUDE_BLUETOOTH_API_PROTOTYPES
   typedef Event_t (BTPSAPI *PFN_BTPS_CreateEvent_t)(Boolean_t CreateSignalled);
#endif

   /* The following function is responsible for waiting for the         */
   /* specified Event to become Signalled.  This function accepts as    */
   /* input the Event Handle to wait for, and the Timeout (specified    */
   /* in Milliseconds) to wait for the Event to become Signalled.  This */
   /* function returns TRUE if the Event was set to the Signalled       */
   /* State (in the Timeout specified) or FALSE if either there was an  */
   /* error OR the Event was not set to the Signalled State in the      */
   /* specified Timeout.  It should be noted that Signalls have a       */
   /* special property in that multiple Threads can be waiting for the  */
   /* Event to become Signalled and ALL calls to BTPS_WaitEvent() will  */
   /* return TRUE whenever the state of the Event becomes Signalled.    */
```

```
BTPSAPI_DECLARATION Boolean_t BTPSAPI BTPS_WaitEvent(Event_t Event, unsigned long
Timeout);

#ifdef INCLUDE_BLUETOOTH_API_PROTOTYPES
   typedef Boolean_t (BTPSAPI *PFN_BTPS_WaitEvent_t)(Event_t Event, unsigned long
Timeout);
#endif

   /* The following function is responsible for changing the state of   */
   /* the specified Event to the Non-Signalled State.  Once the Event   */
   /* is in this State, ALL calls to the BTPS_WaitEvent() function will */
   /* block until the State of the Event is set to the Signalled State. */
   /* This function accepts as input the Event Handle of the Event to   */
   /* set to the Non-Signalled State.                                   */
BTPSAPI_DECLARATION void BTPSAPI BTPS_ResetEvent(Event_t Event);

#ifdef INCLUDE_BLUETOOTH_API_PROTOTYPES
   typedef void (BTPSAPI *PFN_BTPS_ResetEvent_t)(Event_t Event);
#endif

   /* The following function is responsible for changing the state of   */
   /* the specified Event to the Signalled State.  Once the Event is in */
   /* this State, ALL calls to the BTPS_WaitEvent() function will       */
   /* return.  This function accepts as input the Event Handle of the   */
   /* Event to set to the Signalled State.                              */
BTPSAPI_DECLARATION void BTPSAPI BTPS_SetEvent(Event_t Event);

#ifdef INCLUDE_BLUETOOTH_API_PROTOTYPES
   typedef void (BTPSAPI *PFN_BTPS_SetEvent_t)(Event_t Event);
#endif

   /* The following function is responsible for destroying an Event that*/
   /* was created successfully via a successful call to the             */
   /* BTPS_CreateEvent() function.  This function accepts as input the  */
   /* Event Handle of the Event to destroy.  Once this function is      */
   /* completed the Event Handle is NO longer valid and CANNOT be       */
   /* used.  Calling this function will cause all outstanding           */
   /* BTPS_WaitEvent() functions to fail with an error.                 */
BTPSAPI_DECLARATION void BTPSAPI BTPS_CloseEvent(Event_t Event);

#ifdef INCLUDE_BLUETOOTH_API_PROTOTYPES
   typedef void (BTPSAPI *PFN_BTPS_CloseEvent_t)(Event_t Event);
#endif

   /* The following function is provided to allow a mechanism to        */
   /* actually allocate a Block of Memory (of at least the specified    */
   /* size).  This function accepts as input the size (in Bytes) of the */
   /* Block of Memory to be allocated.  This function returns a NON-NULL*/
   /* pointer to this Memory Buffer if the Memory was successfully      */
   /* allocated, or a NULL value if the memory could not be allocated.  */
BTPSAPI_DECLARATION void *BTPSAPI BTPS_AllocateMemory(unsigned long MemorySize);

#ifdef INCLUDE_BLUETOOTH_API_PROTOTYPES
   typedef void *(BTPSAPI *PFN_BTPS_AllocateMemory_t)(unsigned long MemorySize);
#endif

   /* The following function is responsible for de-allocating a Block   */
   /* of Memory that was successfully allocated with the                */
   /* BTPS_AllocateMemory() function.  This function accepts a NON-NULL */
   /* Memory Pointer which was returned from the BTPS_AllocateMemory()  */
   /* function.  After this function completes the caller CANNOT use    */
   /* ANY of the Memory pointed to by the Memory Pointer.               */
BTPSAPI_DECLARATION void BTPSAPI BTPS_FreeMemory(void *MemoryPointer);

#ifdef INCLUDE_BLUETOOTH_API_PROTOTYPES
   typedef void (BTPSAPI *PFN_BTPS_FreeMemory_t)(void *MemoryPointer);
#endif

   /* The following function is responsible for copying a block of      */
   /* memory of the specified size from the specified source pointer    */
   /* to the specified destination memory pointer.  This function       */
   /* accepts as input a pointer to the memory block that is to be      */
   /* Destination Buffer (first parameter), a pointer to memory block   */
   /* that points to the data to be copied into the destination buffer, */
   /* and the size (in bytes) of the Data to copy.  The Source and      */
   /* Destination Memory Buffers must contain AT LEAST as many bytes    */
```

```
   /* as specified by the Size parameter.                              */
   /* * NOTE * This function does not allow the overlapping of the     */
   /*          Source and Destination Buffers !!!!                     */
BTPSAPI_DECLARATION void BTPSAPI BTPS_MemCopy(void *Destination, BTPSCONST void *Source,
unsigned long Size);

#ifdef INCLUDE_BLUETOOTH_API_PROTOTYPES
   typedef void (BTPSAPI *PFN_BTPS_MemCopy_t)(void *Destination, BTPSCONST void *Source,
unsigned long Size);
#endif

   /* The following function is responsible for moving a block of      */
   /* memory of the specified size from the specified source pointer   */
   /* to the specified destination memory pointer.  This function      */
   /* accepts as input a pointer to the memory block that is to be     */
   /* Destination Buffer (first parameter), a pointer to memory block  */
   /* that points to the data to be copied into the destination buffer, */
   /* and the size (in bytes) of the Data to copy.  The Source and     */
   /* Destination Memory Buffers must contain AT LEAST as many bytes   */
   /* as specified by the Size parameter.                              */
   /* * NOTE * This function DOES allow the overlapping of the         */
   /*          Source and Destination Buffers.                         */
BTPSAPI_DECLARATION void BTPSAPI BTPS_MemMove(void *Destination, BTPSCONST void *Source,
unsigned long Size);

#ifdef INCLUDE_BLUETOOTH_API_PROTOTYPES
   typedef void (BTPSAPI *PFN_BTPS_MemMove_t)(void *Destination, BTPSCONST void *Source,
unsigned long Size);
#endif

   /* The following function is provided to allow a mechanism to fill  */
   /* a block of memory with the specified value.  This function accepts*/
   /* as input a pointer to the Data Buffer (first parameter) that is  */
   /* to filled with the specified value (second parameter).  The      */
   /* final parameter to this function specifies the number of bytes   */
   /* that are to be filled in the Data Buffer.  The Destination       */
   /* Buffer must point to a Buffer that is AT LEAST the size of       */
   /* the Size parameter.                                              */
BTPSAPI_DECLARATION void BTPSAPI BTPS_MemInitialize(void *Destination, unsigned char
Value, unsigned long Size);

#ifdef INCLUDE_BLUETOOTH_API_PROTOTYPES
   typedef void (BTPSAPI *PFN_BTPS_MemInitialize_t)(void *Destination, unsigned char
Value, unsigned long Size);
#endif

   /* The following function is provided to allow a mechanism to       */
   /* Compare two blocks of memory to see if the two memory blocks     */
   /* (each of size Size (in bytes)) are equal (each and every byte up  */
   /* to Size bytes).  This function returns a negative number if      */
   /* Source1 is less than Source2, zero if Source1 equals Source2, and */
   /* a positive value if Source1 is greater than Source2.             */
BTPSAPI_DECLARATION int BTPSAPI BTPS_MemCompare(BTPSCONST void *Source1, BTPSCONST void
*Source2, unsigned long Size);

#ifdef INCLUDE_BLUETOOTH_API_PROTOTYPES
   typedef int (BTPSAPI *PFN_BTPS_MemCompare_t)(BTPSCONST void *Source1, BTPSCONST void
*Source2, unsigned long Size);
#endif

   /* The following function is provided to allow a mechanism to Compare*/
   /* two blocks of memory to see if the two memory blocks (each of size*/
   /* Size (in bytes)) are equal (each and every byte up to Size bytes) */
   /* using a Case-Insensitive Compare.  This function returns a       */
   /* negative number if Source1 is less than Source2, zero if Source1  */
   /* equals Source2, and a positive value if Source1 is greater than  */
   /* Source2.                                                         */
BTPSAPI_DECLARATION int BTPSAPI BTPS_MemCompareI(BTPSCONST void *Source1, BTPSCONST void
*Source2, unsigned long Size);

#ifdef INCLUDE_BLUETOOTH_API_PROTOTYPES
   typedef int (BTPSAPI *PFN_BTPS_MemCompareI_t)(BTPSCONST void *Source1, BTPSCONST void
*Source2, unsigned int Size);
#endif

   /* The following function is provided to allow a mechanism to       */
```

```
   /* copy a source NULL Terminated ASCII (character) String to the     */
   /* specified Destination String Buffer.  This function accepts as    */
   /* input a pointer to a buffer (Destination) that is to receive the  */
   /* NULL Terminated ASCII String pointed to by the Source parameter.  */
   /* This function copies the string byte by byte from the Source      */
   /* to the Destination (including the NULL terminator).               */
BTPSAPI_DECLARATION void BTPSAPI BTPS_StringCopy(char *Destination, BTPSCONST char
*Source);

#ifdef INCLUDE_BLUETOOTH_API_PROTOTYPES
   typedef void (BTPSAPI *PFN_BTPS_StringCopy_t)(char *Destination, BTPSCONST char
*Source);
#endif

   /* The following function is provided to allow a mechanism to        */
   /* determine the Length (in characters) of the specified NULL        */
   /* Terminated ASCII (character) String.  This function accepts as    */
   /* input a pointer to a NULL Terminated ASCII String and returns     */
   /* the number of characters present in the string (NOT including     */
   /* the terminating NULL character).                                  */
BTPSAPI_DECLARATION unsigned int BTPSAPI BTPS_StringLength(BTPSCONST char *Source);

#ifdef INCLUDE_BLUETOOTH_API_PROTOTYPES
   typedef unsigned int (BTPSAPI *PFN_BTPS_StringLength_t)(BTPSCONST char *Source);
#endif

   /* The following MACRO definition is provided to allow a mechanism   */
   /* for a C Run-Time Library sprintf() function implementation.  This */
   /* MACRO could be redefined as a function (like the rest of the      */
   /* functions in this file), however more code would be required to   */
   /* implement the variable number of arguments and formatting code    */
   /* then it would be to simply call the C Run-Time Library sprintf()  */
   /* function.  It is simply provided here as a MACRO mapping to allow */
   /* an easy means for a starting place to port this file to other     */
   /* operating systems/platforms.                                      */
#define BTPS_SprintF sprintf

   /* The following function is provided to allow a means for the       */
   /* programmer to create a seperate thread of execution.  This        */
   /* function accepts as input the Function that represents the        */
   /* Thread that is to be installed into the system as its first       */
   /* parameter.  The second parameter is the size of the Threads       */
   /* Stack (in bytes) required by the Thread when it is executing.     */
   /* The final parameter to this function represents a parameter that  */
   /* is to be passed to the Thread when it is created.  This function  */
   /* returns a NON-NULL Thread Handle if the Thread was successfully   */
   /* created, or a NULL Thread Handle if the Thread was unable to be   */
   /* created.  Once the thread is created, the only way for the Thread */
   /* to be removed from the system is for the Thread function to run   */
   /* to completion.                                                    */
   /* * NOTE * There does NOT exist a function to Kill a Thread that    */
   /*          is present in the system.  Because of this, other means  */
   /*          needs to be devised in order to signal the Thread that   */
   /*          it is to terminate.                                      */
BTPSAPI_DECLARATION ThreadHandle_t BTPSAPI BTPS_CreateThread(Thread_t ThreadFunction,
unsigned int StackSize, void *ThreadParameter);

#ifdef INCLUDE_BLUETOOTH_API_PROTOTYPES
   typedef ThreadHandle_t (BTPSAPI *PFN_BTPS_CreateThread_t)(Thread_t ThreadFunction,
unsigned int StackSize, void *ThreadParameter);
#endif

   /* The following function is provided to allow a mechanism to        */
   /* retrieve the handle of the thread which is currently executing.   */
   /* This function require no input parameters and will return a valid */
   /* ThreadHandle upon success.                                        */
BTPSAPI_DECLARATION ThreadHandle_t BTPSAPI BTPS_CurrentThreadHandle(void);

#ifdef INCLUDE_BLUETOOTH_API_PROTOTYPES
   typedef ThreadHandle_t (BTPSAPI *PFN_BTPS_CurrentThreadHandle_t)(void);
#endif

   /* The following function is provided to allow a mechanism to create */
   /* a Mailbox.  A Mailbox is a Data Store that contains slots (all    */
   /* of the same size) that can have data placed into (and retrieved   */
   /* from).  Once Data is placed into a Mailbox (via the               */
```

```
   /* BTPS_AddMailbox() function, it can be retreived by using the      */
   /* BTPS_WaitMailbox() function.  Data placed into the Mailbox is     */
   /* retrieved in a FIFO method.  This function accepts as input the   */
   /* Maximum Number of Slots that will be present in the Mailbox and   */
   /* the Size of each of the Slots.  This function returns a NON-NULL  */
   /* Mailbox Handle if the Mailbox is successfully created, or a       */
   /* NULL Mailbox Handle if the Mailbox was unable to be created.      */
BTPSAPI_DECLARATION Mailbox_t BTPSAPI BTPS_CreateMailbox(unsigned int NumberSlots,
unsigned int SlotSize);

#ifdef INCLUDE_BLUETOOTH_API_PROTOTYPES
   typedef Mailbox_t (BTPSAPI *PFN_BTPS_CreateMailbox_t)(unsigned int NumberSlots,
unsigned int SlotSize);
#endif

   /* The following function is provided to allow a means to Add data   */
   /* to the Mailbox (where it can be retrieved via the                 */
   /* BTPS_WaitMailbox() function.  This function accepts as input the  */
   /* Mailbox Handle of the Mailbox to place the data into and a        */
   /* pointer to a buffer that contains the data to be added.  This     */
   /* pointer *MUST* point to a data buffer that is AT LEAST the Size   */
   /* of the Slots in the Mailbox (specified when the Mailbox was       */
   /* created) and this pointer CANNOT be NULL.  The data that the      */
   /* MailboxData pointer points to is placed into the Mailbox where it */
   /* can be retrieved via the BTPS_WaitMailbox() function.             */
   /* * NOTE * This function copies from the MailboxData Pointer the    */
   /*          first SlotSize Bytes.  The SlotSize was specified when   */
   /*          the Mailbox was created via a successful call to the     */
   /*          BTPS_CreateMailbox() function.                           */
BTPSAPI_DECLARATION Boolean_t BTPSAPI BTPS_AddMailbox(Mailbox_t Mailbox, void
*MailboxData);

#ifdef INCLUDE_BLUETOOTH_API_PROTOTYPES
   typedef Boolean_t (BTPSAPI *PFN_BTPS_AddMailbox_t)(Mailbox_t Mailbox, void
*MailboxData);
#endif

   /* The following function is provided to allow a means to retrieve   */
   /* data from the specified Mailbox.  This function will block until  */
   /* either Data is placed in the Mailbox or an error with the Mailbox */
   /* was detected.  This function accepts as its first parameter a     */
   /* Mailbox Handle that represents the Mailbox to wait for the data   */
   /* with.  This function accepts as its second parameter, a pointer   */
   /* to a data buffer that is AT LEAST the size of a single Slot of    */
   /* the Mailbox (specified when the BTPS_CreateMailbox() function     */
   /* was called).  The MailboxData parameter CANNOT be NULL.  This     */
   /* function will return TRUE if data was successfully retrieved      */
   /* from the Mailbox or FALSE if there was an error retrieving data   */
   /* from the Mailbox.  If this function returns TRUE then the first   */
   /* SlotSize bytes of the MailboxData pointer will contain the data   */
   /* that was retrieved from the Mailbox.                              */
   /* * NOTE * This function copies to the MailboxData Pointer the      */
   /*          data that is present in the Mailbox Slot (of size        */
   /*          SlotSize).  The SlotSize was specified when the Mailbox  */
   /*          was created via a successful call to the                 */
   /*          BTPS_CreateMailbox() function.                           */
BTPSAPI_DECLARATION Boolean_t BTPSAPI BTPS_WaitMailbox(Mailbox_t Mailbox, void
*MailboxData);

#ifdef INCLUDE_BLUETOOTH_API_PROTOTYPES
   typedef Boolean_t (BTPSAPI *PFN_BTPS_WaitMailbox_t)(Mailbox_t Mailbox, void
*MailboxData);
#endif

   /* The following function is responsible for destroying a Mailbox    */
   /* that was created successfully via a successful call to the        */
   /* BTPS_CreateMailbox() function.  This function accepts as input    */
   /* the Mailbox Handle of the Mailbox to destroy.  Once this function */
   /* is completed the Mailbox Handle is NO longer valid and CANNOT be  */
   /* used.  Calling this function will cause all outstanding           */
   /* BTPS_WaitMailbox() functions to fail with an error.               */
BTPSAPI_DECLARATION void BTPSAPI BTPS_DeleteMailbox(Mailbox_t Mailbox);

#ifdef INCLUDE_BLUETOOTH_API_PROTOTYPES
   typedef void (BTPSAPI *PFN_BTPS_DeleteMailbox_t)(Mailbox_t Mailbox);
#endif
```

```
   /* The following function is used to initialize the Platform module. */
   /* The Platform module relies on some static variables that are used */
   /* to coordinate the abstraction.  When the module is initially      */
   /* started from a cold boot, all variables are set to the proper     */
   /* state.  If the Warm Boot is required, then these variables need to*/
   /* be reset to their default values.  This function sets all static  */
   /* parameters to their default values.                               */
   /* * NOTE * The implementation is free to pass whatever information  */
   /*          required in this parameter.                              */
BTPSAPI_DECLARATION void BTPSAPI BTPS_Init(void *UserParam);

#ifdef INCLUDE_BLUETOOTH_API_PROTOTYPES
   typedef void (BTPSAPI *PFN_BTPS_Init_t)(void *UserParam);
#endif

   /* The following function is used to cleanup the Platform module.    */
BTPSAPI_DECLARATION void BTPSAPI BTPS_DeInit(void);

#ifdef INCLUDE_BLUETOOTH_API_PROTOTYPES
   typedef void (BTPSAPI *PFN_BTPS_DeInit_t)(void);
#endif

   /* Write out the specified NULL terminated Debugging String to the   */
   /* Debug output.                                                     */
BTPSAPI_DECLARATION void BTPSAPI BTPS_OutputMessage(BTPSCONST char *DebugString, ...);

#ifdef INCLUDE_BLUETOOTH_API_PROTOTYPES
   typedef void (BTPSAPI *PFN_BTPS_OutputMessage_t)(BTPSCONST char *DebugString, ...);
#endif

   /* The following function is used to set the Debug Mask that controls*/
   /* which debug zone messages get displayed.  The function takes as   */
   /* its only parameter the Debug Mask value that is to be used.  Each */
   /* bit in the mask corresponds to a debug zone.  When a bit is set,  */
   /* the printing of that debug zone is enabled.                       */
BTPSAPI_DECLARATION void BTPSAPI BTPS_SetDebugMask(unsigned long DebugMask);

#ifdef INCLUDE_BLUETOOTH_API_PROTOTYPES
   typedef void (BTPSAPI *PFN_BTPS_SetDebugMask_t)(unsigned long DebugMask);
#endif

   /* The following function is a utility function that can be used to  */
   /* determine if a specified Zone is currently enabled for debugging. */
BTPSAPI_DECLARATION int BTPSAPI BTPS_TestDebugZone(unsigned long Zone);

#ifdef INCLUDE_BLUETOOTH_API_PROTOTYPES
   typedef int (BTPSAPI *PFN_BTPS_TestDebugZone_t)(unsigned long Zone);
#endif

   /* The following function is responsible for writing binary debug    */
   /* data to the specified debug file handle.  The first parameter to  */
   /* this function is the handle of the open debug file to write the   */
   /* debug data to.  The second parameter to this function is the      */
   /* length of binary data pointed to by the next parameter.  The final*/
   /* parameter to this function is a pointer to the binary data to be  */
   /* written to the debug file.                                        */
BTPSAPI_DECLARATION int BTPSAPI BTPS_DumpData(unsigned int DataLength, BTPSCONST unsigned
char *DataPtr);

#ifdef INCLUDE_BLUETOOTH_API_PROTOTYPES
   typedef int (BTPSAPI *PFN_BTPS_DumpData_t)(unsigned int DataLength, BTPSCONST unsigned
char *DataPtr);
#endif

#endif
```