Bluetooth Protocol Stack Kernel

(Multi-threaded O/S)

Application Programming Interface Reference Manual

Release: 2.1.3
January 19, 2011

Louisville, KY www.stonestreetone.com

Table Of Contents

1. Introduction

Bluetopia®, the Bluetooth Protocol Stack by Stonestreet One provides a software architecture that encapsulates the upper functionality of the Bluetooth Protocol Stack. More specifically, this stack is a software solution that resides above the Physical HCI (Host Controller Interface) Transport Layer and extends through the L2CAP (Logical Link Control and Adaptation Protocol) and the SCO (Synchronous Connection-Oriented) Link layers. In addition to basic functionality at these layers, the Bluetooth Protocol Stack by Stonestreet One provides implementations of the Service Discovery Protocol (SDP), RFCOMM (the Radio Frequency serial COMMunications port emulator), and several of the Bluetooth Profiles. Program access to these layers, services, and profiles is handled via Application Programming Interface (API) calls.

This document focuses on the API reference that contains a description of all programming interfaces for Stonestreet One's Bluetooth Protocol Stack Kernel, which can be used to abstract the actual operating system API/functionality. The programmer is free to use the API defined in this document or use the native O/S API. It should be noted, however, that these functions must contain code because these are the entry points that Bluetopia® uses when it executes.

1.1 Scope

This reference manual provides information on the Bluetooth Protocol Stack Kernel API.

1.2 Applicable Documents

The following documents may be used for additional background and technical depth regarding the Bluetooth technology.

1. *Specification of the Bluetooth System, Volume 0, Master Table of Contents & Compliance Requirements*, version 2.1+EDR, July 26, 2007.
2. *Specification of the Bluetooth System, Volume 1, Architecture and Terminology Overview*, version 2.1+EDR, July 26, 2007.
3. *Specification of the Bluetooth System, Volume 2, Core System Package [Controller Volume]*, version 2.1+EDR, July 26, 2007.
4. *Specification of the Bluetooth System, Volume 3, Core System Package [Host Volume]*, version 2.1+EDR, July 26, 2007.
5. *Specification of the Bluetooth System, Volume 4, Host Controller Interface*, version 2.1+EDR, July 26, 2007.
6. *Specification of the Bluetooth System, Bluetooth Core Specification Addendum 1*, June 26, 2008.
7. *Bluetopia® Protocol Stack, System Call Requirements,* version 1.7, July 14, 2009
8. *Bluetopia® Protocol Stack, Application Programming Interface Reference Manual,* version 2.0, July 1, 2008.

1.3 Acronyms and Abbreviations

Acronyms and abbreviations used in this document and other Bluetooth specifications are listed in the table below.

Term	Meaning
API	Application Programming Interface
BD_ADDR	Bluetooth Device Address
BT	Bluetooth
BTPS	Bluetooth Protocol Stack
FIFO	First In First Out
LSB	Least Significant Bit
MSB	Most Significant Bit
UART	Universal Asynchronous Receiver/Transmitter
USB	Universal Serial Bus

2. Bluetooth Protocol Stack Kernel Programming Interface

The Bluetooth Protocol Stack Kernel programming interface defines the procedures to be used to when using the Stonestreet One Bluetooth Protocol Stack Kernel. The Bluetooth Protocol Stack Kernel commands are listed in section 2.1 and the prototype for the Thread Function is described in section 2.2. The actual prototypes and constants outlined in this section can be found in the **BKRNLAPI.H** header file in the Bluetopia distribution.

2.1 Bluetooth Protocol Stack Kernel Commands

The available Bluetooth Protocol Stack Kernel command functions are listed in the table below and are described in the text that follows.

Function	Description
BTPS_Delay	Delay the current task for the time specified.
BTPS_CreateMutex	Create a mutex (binary semaphore).
BTPS_WaitMutex	Wait for a specified mutex to become free.
BTPS_ReleaseMutex	Release an acquired mutex.
BTPS_CloseMutex	Destroy a mutex.
BTPS_CreateEvent	Create an event.
BTPS_WaitEvent	Wait for a specified event to become signalled.
BTPS_ResetEvent	Change an event to the non-signalled state.
BTPS_SetEvent	Change an event to the signalled state.
BTPS_CloseEvent	Destroy a created previously created event.
BTPS_AllocateMemory	Allocate a block of memory.
BTPS_FreeMemory	Free (de-allocate) a previously allocated block of memory.
BTPS_MemCopy	Copy a block of memory from a source to a destination.
BTPS_MemMove	Copy a block of memory from a source to a destination.
BTPS_MemInitialize	Fill a block of memory with a specified value.
BTPS_MemCompare	Compare two blocks of memory to see if they are equal.

BTPS_MemCompareI	Compare two blocks of memory to see if they are equal using a case insensitive compare.
BTPS_StringCopy	Copy a NULL terminated ASCII string to a destination.
BTPS_StringLength	Determine the length of a NULL terminated ASCII string.
BTPS_SprintF	Macro mapping of C run-time library sprintf() function.
BTPS_CreateThread	Create a separate thread of execution.
BTPS_CurrentThreadHandle	Determine the context of the current thread of execution.
BTPS_CreateMailbox	Create a mailbox.
BTPS_AddMailbox	Add data to a mailbox.
BTPS_WaitMailbox	Retrieve data from a mailbox.
BTPS_DeleteMailbox	Delete a mailbox.
BTPS_Init	Module initialization function.
BTPS_DeInit	Module de-initialization function.
BTPS_OutputMessage	Output debugging function (not currently called by Bluetopia).
BTPS_SetDebugMask	Update the current Debugging Zone Mask.
BTPS_TestDebugMask	Determine if a specified Debug Zone Mask is enabled.
BTPS_DumpData	Output debugging function (not currently called by Bluetopia).

BTPS_Delay .

The following function is responsible for delaying the currently executing thread for the specified duration (specified in milliseconds). Very small timeouts might be smaller in granularity than the system can support.

Prototype:

void BTPSAPI **BTPS_Delay**(unsigned long MilliSeconds)

Parameters:

MilliSeconds	Number of milliseconds to delay.

Return:

BTPS_CreateMutex

The following function is responsible for creating a 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 contrast to a semaphore that will block waiting for the second acquisition of the semaphore. If a Mutex is successfully created, it can only be destroyed by calling the **BTPS_CloseMutex**() function (and passing the returned Mutex Handle).

Prototype:

Mutex_t BTPSAPI **BTPS_CreateMutex**(Boolean_t CreateOwned)

Parameters:

CreateOwned	Boolean to indicate whether or not the Mutex is initially signalled. If this parameter is TRUE, the caller owns the Mutex and any other threads waiting on the Mutex will block. If this parameter is FALSE, the Mutex is created un-owned, which means that the caller does not own the Mutex. The Mutex, at this point is able to be acquired by a call to **BTPS_WaitMutex**().

Return:

NON-NULL Mutex Handle if the Mutex was successfully created.

NULL Mutex Handle if the Mutex was unable to be created.

BTPS_WaitMutex

The following function is responsible for waiting for the specified Mutex to become free. 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. There must exist a corresponding **BTPS_ReleaseMutex**() function call for EVERY successful **BTPS_WaitMutex**() function call or a deadlock will occur in the system.

Prototype:

Boolean_t BTPSAPI **BTPS_WaitMutex**(Mutex_t Mutex, unsigned long Timeout)

Parameters:

Mutex	Mutex Handle to wait for.

Timeout	Time to wait for the Mutex to become available, specified in milliseconds. An infinite timeout value (wait forever) can be specified by using the following constant: BTPS_INFINITE_WAIT

Return:

TRUE (non-zero) if the Mutex was successfully acquired.

FALSE (zero) if either there was an error or the Mutex was not acquired in the specified timeout period.

BTPS_ReleaseMutex .

The following function is responsible for releasing a Mutex that was successfully acquired with the **BTPS_WaitMutex**() function. There must exist a corresponding **BTPS_ReleaseMutex**() function call for every successful **BTPS_WaitMutex**() function call or a deadlock will occur in the system.

Prototype:

void BTPSAPI **BTPS_ReleaseMutex**(Mutex_t Mutex)

Parameters:

Mutex	Handle of currently owned Mutex. This Mutex must have been acquired using the **BTPS_WaitMutex**() function.

Return:

BTPS_CloseMutex .

The following function is responsible for destroying a Mutex that was created successfully via a successful call to the **BTPS_CreateMutex**() function. 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.

Prototype:

void BTPSAPI **BTPS_CloseMutex**(Mutex_t Mutex)

Parameters:

Mutex	Handle of Mutex to be destroyed. This value was returned via a successful call to the **BTPS_CreateMutex**() function.

Return:

BTPS_CreateEvent

The following function is responsible for creating an 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. If an Event is successfully created, it can only be destroyed by calling the **BTPS_CloseEvent**() function (and passing the returned Event Handle).

Prototype:

Event_t BTPSAPI **BTPS_CreateEvent**(Boolean_t Signalled)

Parameters:

Signalled	Boolean to indicate if the Event is initially signalled or not. If this parameter is TRUE, the state of the Event is signalled and any **BTPS_WaitEvent**() function call will immediately return. If this parameter is FALSE, the state of the Event is non-signalled.

Return:

NON-NULL Event Handle if the Event was successfully created.

NULL Event Handle if the Event was unable to be created.

BTPS_WaitEvent

The following function is responsible for waiting for the specified Event to become signalled. It should be noted that signals 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.

Prototype:

Boolean_t BTPSAPI **BTPS_WaitEvent**(Event_t Event, unsigned long Timeout)

Parameters:

Event	Handle of the Event to wait for
Timeout	Time to wait for the Event to become Signalled, specified in milliseconds. An infinite timeout value (wait forever) can be specified by using the following constant: BTPS_INFINITE_WAIT

Return:

TRUE (non-zero) if the Event was set to the signalled state in the timeout period specified.

FALSE (zero) if either there was an error or the Event was not set to the signalled state in the specified timeout period.

BTPS_ResetEvent

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.

Prototype:

void BTPSAPI **BTPS_ResetEvent**(Event_t Event)

Parameters:

Event	Event Handle of the Event to set to the non-signalled state.

Return:

BTPS_SetEvent

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.

Prototype:

void BTPSAPI **BTPS_SetEvent**(Event_t Event)

Parameters:

Event	Event Handle of the Event to set to the signalled state.

Return:

BTPS_CloseEvent

The following function is responsible for destroying an Event that was created via a successful call to the **BTPS_CreateEvent**() function. Once this function is complete, 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.

Prototype:

void BTPSAPI **BTPS_CloseEvent**(Event_t Event)

Parameters:

Event	Event Handle of the Event to destroy. This handle was obtained via a successful call to the **BTPS_CreateEvent**() function.

Return:

BTPS_AllocateMemory

The following function is provided to allow a mechanism to actually allocate a block of memory (of at least the specified size). The memory can later be returned to the system by calling the **BTPS_FreeMemory**() function.

Prototype:

void *BTPSAPI **BTPS_AllocateMemory**(unsigned int MemorySize)

Parameters:

MemorySize	The size (in bytes) of the block of memory to be allocated.

Return:

NON-NULL pointer to this memory buffer if the memory was successfully allocated.

NULL pointer if the memory could not be allocated.

BTPS_FreeMemory .

The following function is responsible for de-allocating a block of memory that was successfully allocated with the **BTPS_AllocateMemory**() function. After this function completes the caller CANNOT use ANY of the memory pointed to by the memory pointer that was freed by this function.

Prototype:

void BTPSAPI **BTPS_FreeMemory**(void *MemoryPointer)

Parameters:

MemoryPointer	A NON-NULL memory pointer which was returned from the **BTPS_AllocateMemory**() function.

Return:

BTPS_MemCopy .

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. The source and destination memory buffers must contain AT LEAST as many bytes as specified by the Size parameter. This function does not allow the overlapping of the Source and Destination Buffers.

Prototype:

void BTPSAPI **BTPS_MemCopy**(void *Destination, void *Source, unsigned int Size)

Parameters:

Destination	A pointer to the memory block that is to be destination buffer.
Source	A pointer to the source memory block that points to the data to be copied into the destination buffer.
Size	The size, in bytes, of the data to copy.

Return:

BTPS_MemMove

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. The source and destination memory buffers must contain AT LEAST as many bytes as specified by the Size parameter. This function DOES allow the overlapping of the Source and Destination Buffers.

Prototype:

void BTPSAPI **BTPS_MemCopy**(void *Destination, void *Source, unsigned int Size)

Parameters:

Destination	A pointer to the memory block that is to be destination buffer.
Source	A pointer to the source memory block that points to the data to be copied into the destination buffer.
Size	The size, in bytes, of the data to copy.

Return:

BTPS_MemInitialize

The following function is provided to allow a mechanism to fill a block of memory with the specified value. The destination buffer must point to a buffer that is AT LEAST the size of the Size parameter.

Prototype:

void BTPSAPI **BTPS_MemInitialize**(void *Destination, unsigned char Value, unsigned int Size)

Parameters:

Destination	A pointer to the data buffer that is to be filled with the specified value.
Value	The value that is to be filled into the data buffer.
Size	The number of bytes that are to be filled in the data buffer.

Return:

BTPS_MemCompare

The following function is provided to allow a mechanism to compare two blocks of memory to see if the two memory blocks (each of the size specified by the Size parameter (in bytes)) are equal (each and every byte up to Size bytes).

Prototype:

int BTPSAPI **BTPS_MemCompare**(void *Source1, void *Source2, unsigned int Size)

Parameters:

Source1	A pointer to the first block of memory to be compared.
Source2	A pointer to the second block of memory to be compared.
Size	Number of bytes to compare.

Return:

Negative value if Source1 is less than Source2.

Zero if Source1 equals Source2.

Positive value if Source1 is greater than Source2.

BTPS_MemCompareI .

The following function is provided to allow a mechanism to compare two blocks of memory to see if the two memory blocks (each of the size specified by the Size parameter (in bytes)) are equal (each and every byte up to Size bytes) using a case-insensitive compare.

Prototype:

int BTPSAPI **BTPS_MemCompareI**(void *Source1, void *Source2, unsigned int Size)

Parameters:

Source1	A pointer to the first block of memory to be compared.
Source2	A pointer to the second block of memory to be compared.
Size	Number of bytes to compare.

Return:

Negative value if Source1 is less than Source2.

Zero if Source1 equals Source2.

Positive value if Source1 is greater than Source2.

BTPS_StringCopy .

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 copies the string byte by byte from the source to the destination (including the NULL terminator).

Prototype:

void BTPSAPI **BTPS_StringCopy**(char *Destination, char *Source)

Parameters:

Destination	A pointer to a buffer that is to receive the NULL terminated ASCII string pointed to by the Source parameter.

Source	A pointer to a NULL Terminated ASCII string source buffer that is copied into the buffer pointed to by the destination parameter.

Return:

BTPS_StringLength

The following function is provided to allow a mechanism to determine the length (in character bytes) of the specified NULL terminated ASCII (character) string.

Prototype:

unsigned int BTPSAPI **BTPS_StringLength**(char *Source)

Parameters:

Source	A pointer to a NULL terminated ASCII string.

Return:

The number of characters present in the string (NOT including the terminating NULL character)

BTPS_SprintF

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.

Prototype:

#define **BTPS_SprintF** sprintf

Parameters:

Return:

The number of characters that were written into the output string (not counting the NULL terminator).

BTPS_CreateThread

The following function is provided to allow a means for the programmer to create a separate thread of execution. 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. There does not exist a function to terminate a Thread that is presently executing in the system. Because of this, other means need to be devised in order to signal the Thread that it is to terminate.

Prototype:

ThreadHandle_t BTPSAPI **BTPS_CreateThread**(Thread_t ThreadFunction,
unsigned int StackSize, void *ThreadParameter)

Parameters:

ThreadFunction	Function that represents the Thread that is to be installed into the system.
StackSize	Size of the Thread's Stack (in bytes) required by the Thread when it is executing.
ThreadParameter	Parameter that is to be passed to the Thread when it is created.

Return:

NON-NULL Thread Handle if the Thread was successfully created.

NULL Thread Handle if the Thread was unable to be created.

BTPS_CurrentThreadHandle

The following function is provided to allow a means for the programmer to determine the context of the current thread of execution. This function will return the Thread Handle of the current Thread that is executing. The caller can match this value to either the return value from the **BTPS_CreateThread**() function or a stored Thread Handle value to determine what Thread Context is currently executing.

Prototype:

ThreadHandle_t BTPSAPI **BTPS_CurrentThreadHandle**(void)

Parameters:

Return:

NON-NULL Thread Handle if the Thread Handle was successfully determined.

NULL Thread Handle if the current Thread Handle was unable to be determined.

BTPS_CreateMailbox

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 so that the data can be retrieved at a future time. Once data is placed into a Mailbox (via the **BTPS_AddMailbox**() function), it can be retrieved by using the **BTPS_WaitMailbox**() function. Data placed into the Mailbox is retrieved in a first in first out (FIFO) method.

Prototype:

Mailbox_t BTPSAPI **BTPS_CreateMailbox**(unsigned int NumberSlots,
unsigned int SlotSize)

Parameters:

NumberSlots	The maximum number of slots that will be present in the Mailbox.
SlotSize	Size of each of the slots, in bytes.

Return:

NON-NULL Mailbox Handle if the Mailbox is successfully created.

NULL Mailbox Handle if the Mailbox was unable to be created.

BTPS_AddMailbox .

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. The MailboxData pointer **MUST** point to a data buffer that is AT LEAST the size (in bytes) of a single Slot 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. This function copies from the MailboxData Pointer the first SlotSize bytes. The slot size was specified when the Mailbox was created via a successful call to the **BTPS_CreateMailbox**() function.

Prototype:

Boolean_t BTPSAPI **BTPS_AddMailbox**(Mailbox_t Mailbox, void *MailboxData)

Parameters:

Mailbox	Mailbox Handle of the Mailbox to place the data into.
MailboxData	A pointer to a buffer that contains the data to be added.

Return:

TRUE (non-zero) if successful.

FALSE (zero) if an error occurred.

BTPS_WaitMailbox .

The following function is provided to allow a means to retrieve data from the specified Mailbox. This function will return immediately if either data is placed in the Mailbox or there is no data present in the Mailbox. The MailboxData pointer points 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. If this function returns TRUE then the first SlotSize bytes of the MailboxData pointer will contain the data that was retrieved from the Mailbox. This function copies to the MailboxData Pointer the data that is present in the Mailbox Slot (of size SlotSize). The slot size was specified when the Mailbox was created via a successful call to the **BTPS_CreateMailbox**() function.

Prototype:

Boolean_t BTPSAPI **BTPS_WaitMailbox**(Mailbox_t Mailbox, void *MailboxData)

Parameters:

Mailbox	Mailbox Handle that represents the Mailbox to be used to wait for the data.
MailboxData	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).

Return:

TRUE (non-zero) if data was successfully retrieved from the Mailbox.

FALSE (zero) if there was no Data retrieved from the Mailbox.

BTPS_DeleteMailbox

The following function is responsible for destroying a Mailbox that was created successfully via a successful call to the **BTPS_CreateMailbox**() function. Once this function is completed the Mailbox Handle is NO longer valid and CANNOT be used.

Prototype:

void BTPSAPI **BTPS_DeleteMailbox**(Mailbox_t Mailbox)

Parameters:

Mailbox	Mailbox Handle of the Mailbox to destroy.

Return:

BTPS_Init

This optional function allows for any initialization code specific to a platform.

Prototype:

void BTPSAPI **BTPS_Init**(void *UserParam)

Parameters:

UserParam	Any user required parameter to facilitate system specific initialization.

Return:

BTPS_DeInit

This optional function allows for any de-initialization code specific to a platform.

Prototype:

void BTPSAPI **BTPS_DeInit**(void)

Parameters:

None

Return:

BTPS_OutputMessage .

This optional function allows support for displaying or storing in a file support or debugging information during run-time. A null function must be implemented to support correct operation.

Prototype:

void BTPSAPI **BTPS_ OutputMessage**(char *DebugString, ...)

Parameters:

DebugString	Character string with optional additional arguments to create a text string for display.

Return:

BTPS_SetDebugMask .

This optional function allows support for control of displaying or storing support or debugging information during run-time with different levels of detail. A null function must be implemented to support correct operation.

Prototype:

void BTPSAPI **BTPS_SetDebugMask** (unsigned long DebugMask)

Parameters:

DebugMask	Bit Mask used to control which Debug messages are displayed.

Return:

BTPS_TestDebugZone .

This optional function allows support to determine if a specific execution zone is currently enabled for debugging. A null function must be implemented to support correct operation.

Prototype:

void BTPSAPI **BTPS_TestDebugZone** (unsigned long Zone)

Parameters:

Zone	Bit Mask used to check if a zone is enabled for displaying messages.

Return:

BTPS_DumpData .

This optional function allows displaying binary data in a memory dump format, if the optional display functions are implemented, and if the specific code zones are enabled enabled for debugging. A null function must be implemented to support correct operation.

Prototype:

void BTPSAPI **BTPS_DumpData** (unsigned int DataLength, unsigned char *DataPtr)

Parameters:

DataLength	The length of data to be formatted for display.
DataPtr	A pointer to the data to be formatted for display.

Return:

2.2 BTPS Kernel Thread Function Prototype

The following represents the prototype for a thread function. This function represents the Thread function that will be executed when passed to the **BTPS_CreateThread**() function. Once a thread is created there is no way to kill it. The thread must exit by itself.

Thread_t .

Prototype of function to be added to the scheduler.

Prototype:

void *(BTPSAPI *Thread_t)(void *ThreadParameter);

Parameters:

ThreadParameter	Used defined parameter that is passed to this Thread via the **BTPS_CreateThread**() function.

Return:

Return value is currently ignored and can be anything.

3. File Distributions

The header filse that are distributed with the Bluetooth Protocol Stack Kernel Library is listed in the table below.

File	Contents/Description
BTPSKRNL.h	Bluetooth Protocol Stack Kernel include file.
BKRNLAPI.h	Actual Bluetooth Protocol Stack Kernel API definitions file.

3. 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

#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
```