EtherMind

Porting Guide

8 December 2008
MindTree Consulting Private Limited,
#42, 27th Cross, Banashankari II Stage
Bangalore - 560 070
www.mindtree.com

ABSTRACT

This document describes guidelines for porting EtherMind Bluetooth Stack to new operating systems and/or platforms.

REVISION HISTORY

Owner contact: ethermind_support@mindtree.com

Date	Version	Description	Author
16 August 2004	0.1	Initial Version.	Subhankar Saha
19 August 2004	0.2	Incorporated preliminary review comments. To be added: BT_fops & SM Storage.	Subhankar Saha
02 September 2004	1.0	Incorporated Sharmila's review comments. To be added: BT_fops & SM Storage.	Subhankar Saha
04 January 2005	1.1	Added porting guide for SM Storage	Subhankar Saha
12 January 2005	1.2	Updated BT_thread_detach() has no OUT parameter bt_uart_init() is corrected to uart_bt_init() [Enh ID 653]	Subhankar Saha
12 February 2005	1.3	Porting guideline for BT_fops added. [Enh ID 640]	Anindya Bakshi
03 February 2006	1.4	Added note in the documentation of BT_fops_file_read() & BT_fops_file_write() APIs to clarify that "length" parameter can have a value of Zero as a valid value and should not return Failure. [Enh ID 1209]	Subhankar Saha

Table of Contents

Introduction..... 6

An Overview of EtherMind Stack..... 7

Task Architecture of EtherMind Stack..... 9

Read, Write & Timer Tasks..... 9

Two-Task vs. Single-Task Model..... 10

Things to know before porting..... 11

Source Code Directory Structure..... 11

Operating System and Platform Interfaces..... 12

Porting need of other EtherMind modules..... 13

Porting EtherMind OS Abstraction Module..... 13

Porting OS Abstraction Data Types..... 13

Basic Data Types..... 14

Task/Thread Related Data Types..... 14

Porting OS Abstraction Primitives..... 16

BT_thread_create..... 17

BT_thread_attr_init..... 18

BT_thread_detach..... 18

BT_thread_mutex_init..... 19

BT_thread_mutex_lock..... 20

BT_thread_mutex_unlock..... 20

BT_thread_cond_init..... 21

BT_thread_cond_wait..... 22

BT_thread_cond_signal..... 22

BT_alloc_mem..... 23

BT_free_mem..... 23

BT_mem_copy..... 24

BT_sleep..... 25

BT_usleep..... 25
Customizing Module Initialization & Shutdown..... 26
Porting Serial Port Configuration Module..... 27
Porting HCI UART Transport Module..... 29
Porting HCI-UART Transport Routines..... 29
uart_set_serial_params..... 29
uart_init..... 30
uart_bt_init..... 30
uart_bt_shutdown..... 31
uart_line_read..... 31
uart_send_data..... 32
uart_write_data..... 33
Porting HCI USB Transport Module..... 34
Porting EtherMind Read Task..... 34
Porting EtherMind Write Task..... 35
Porting EtherMind Timer Library..... 36
Porting Needs of EtherMind Timer Library with Timer Task..... 36
The Timer Task..... 36
Getting Current System Time..... 37
Porting Security Manager Persistent Storage..... 37
Porting EtherMind File Operations Abstraction..... 40
Porting File Abstraction Data Types..... 40
File System Related Data Types..... 41
Porting File Abstraction Primitives..... 41
BT_fops_get_current_directory..... 41
BT_fops_get_file_attributes..... 42
BT_fops_set_path_forward..... 42
BT_fops_set_path_backward..... 43

BT_fops_create_folder..... 43
BT_fops_file_open..... 44
BT_fops_file_write..... 45
BT_fops_file_read..... 45
BT_fops_file_close..... 46
BT_fops_object_delete..... 47
Porting EtherMind Debug Library..... 47
References 48

Introduction

The EtherMind Bluetooth Host Protocol stack is a highly portable stack, and till date, it has been ported to numerous operating systems and platforms, as summarized below:

- Operating Systems
    - Embedded – uCLinux, eCOS, uCOS/II, uITRON, VxWorks, WinCE (PocketPC), Nucleus
    - Windows 98/2000/NT/XP – User & Kernel mode
    - Linux – User & Kernel mode
- Processors – x86, MIPS, ARM7, ARM9
- Single Processor Implementation – SiW 1750, SiW 3000, SiW 3500, MindTree Baseband, EPSON
- Transport Interfaces – UART, USB, BCSP and SDIO

To achieve true platform independence for the core protocol and profile modules, several important measures have been taken. This document is intended to act as guidance when the EtherMind stack is required to be ported to a new operating system and/or platform.

The actual changes needed to be done for a particular platform is out of the scope for this document. Instead, this document highlights the key design decision and considerations, including important source code organization of various modules, which has helped tremendously in achieving platform independence.

An Overview of EtherMind Stack

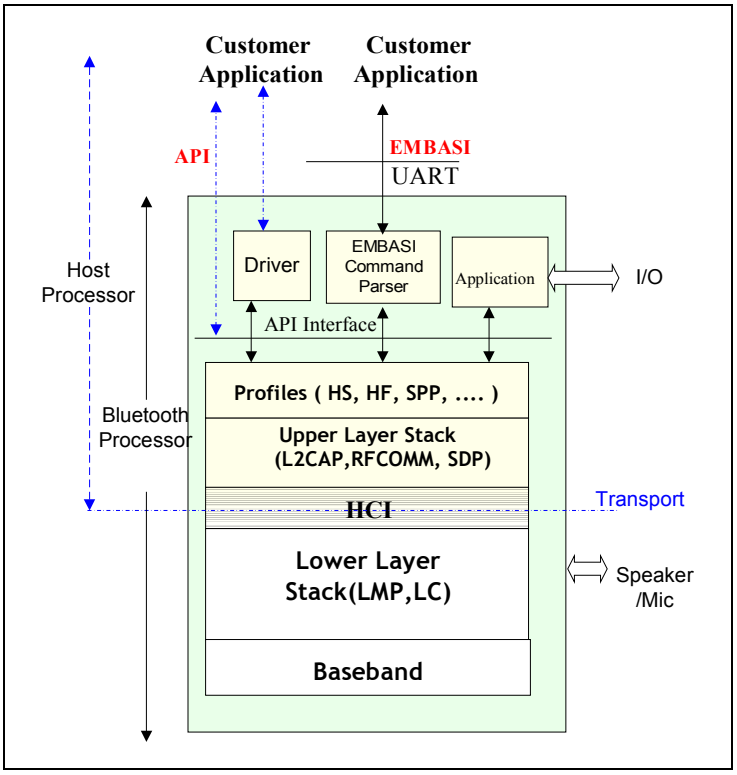
A short description is provided for the EtherMind Stack Architecture to enable the reader with knowledge of certain important facts and information useful while porting the stack onto new operating system and/or platforms.

EtherMind Bluetooth Stack is available as a Host-based Configuration or an Embedded (single processor) Configuration.

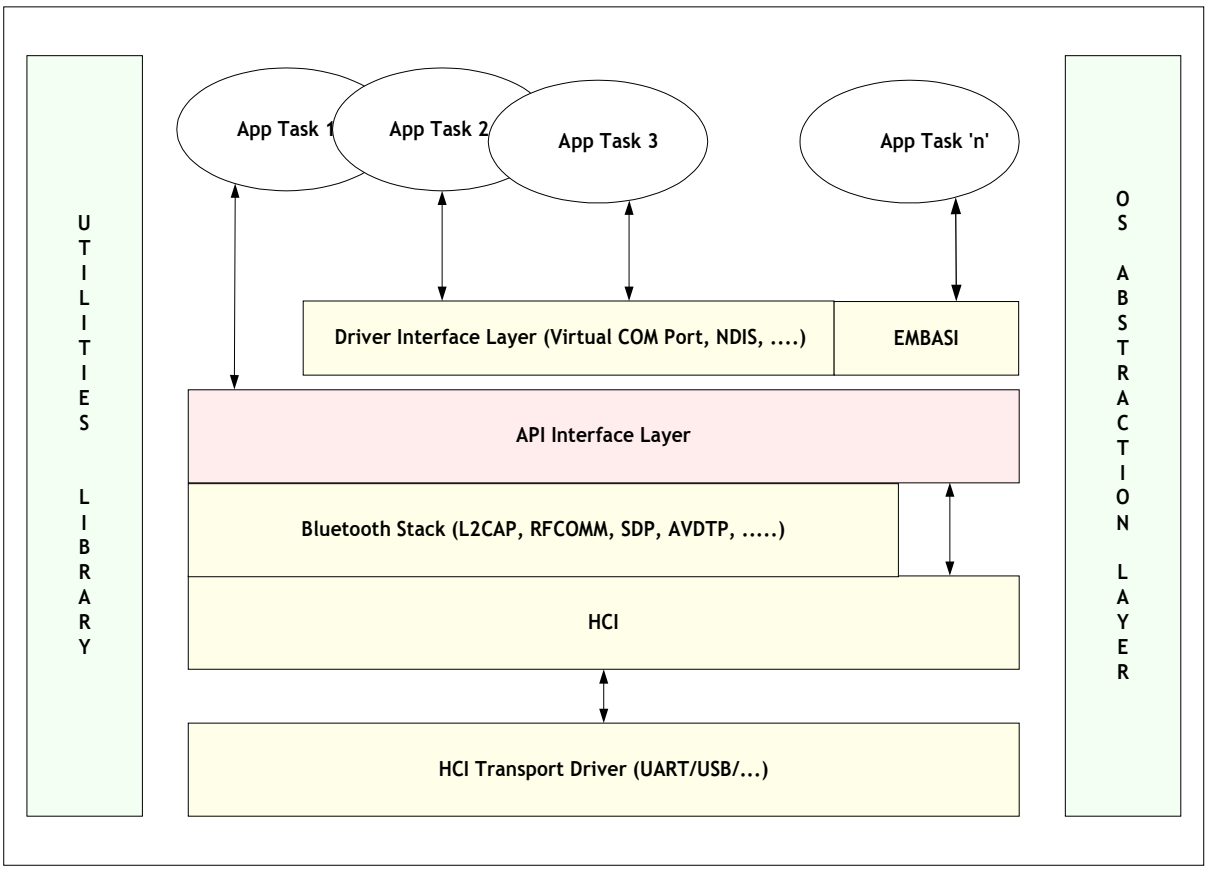
In a Host-based Configuration, the Bluetooth Host Software stack (HCI upwards), profiles and probably the applications run on a Host Processor (application processor), which is distinct from the Bluetooth Processor. The Bluetooth Processor runs the Bluetooth Baseband Controller, Link Manager Protocol (LMP) and HCI only. The stack, profiles and applications running on the Host Processor communicate with the Bluetooth processor, using well-defined transport interfaces (e.g. UART, USB etc.).

In an Embedded Configuration, the Bluetooth Host Software stack (HCI upwards) and profiles reside on the Bluetooth Processor itself. This helps to utilize the (extra) computing power available on the Bluetooth Processor, as well as reduce the cost of the overall solution. The applications can reside on the Bluetooth Processor or an external Application Processor.

A generic block diagram, demonstrating the above concept is shown in the figure below.



The generic architecture of the EtherMind Stack and Profiles is shown in the figure below. Multiple applications can reside on top of the stack. The protocols and profiles of the stack expose a set of APIs, which can be called by the applications. The application tasks can interact with the stack independently, using any of the exported interfaces.



The profiles are independent modules, which use the interfaces exported by the protocols of the stack and in turn export interfaces for the applications.

The APIs exported can be configured to be BLOCKING or NON-BLOCKING using a compilation time flag, while building the stack. The choice of using BLOCKING Vs NON-BLOCKING API varies from product to product and application scenarios. However, all internal interfaces between various stack modules/protocols are strictly Non-Blocking.

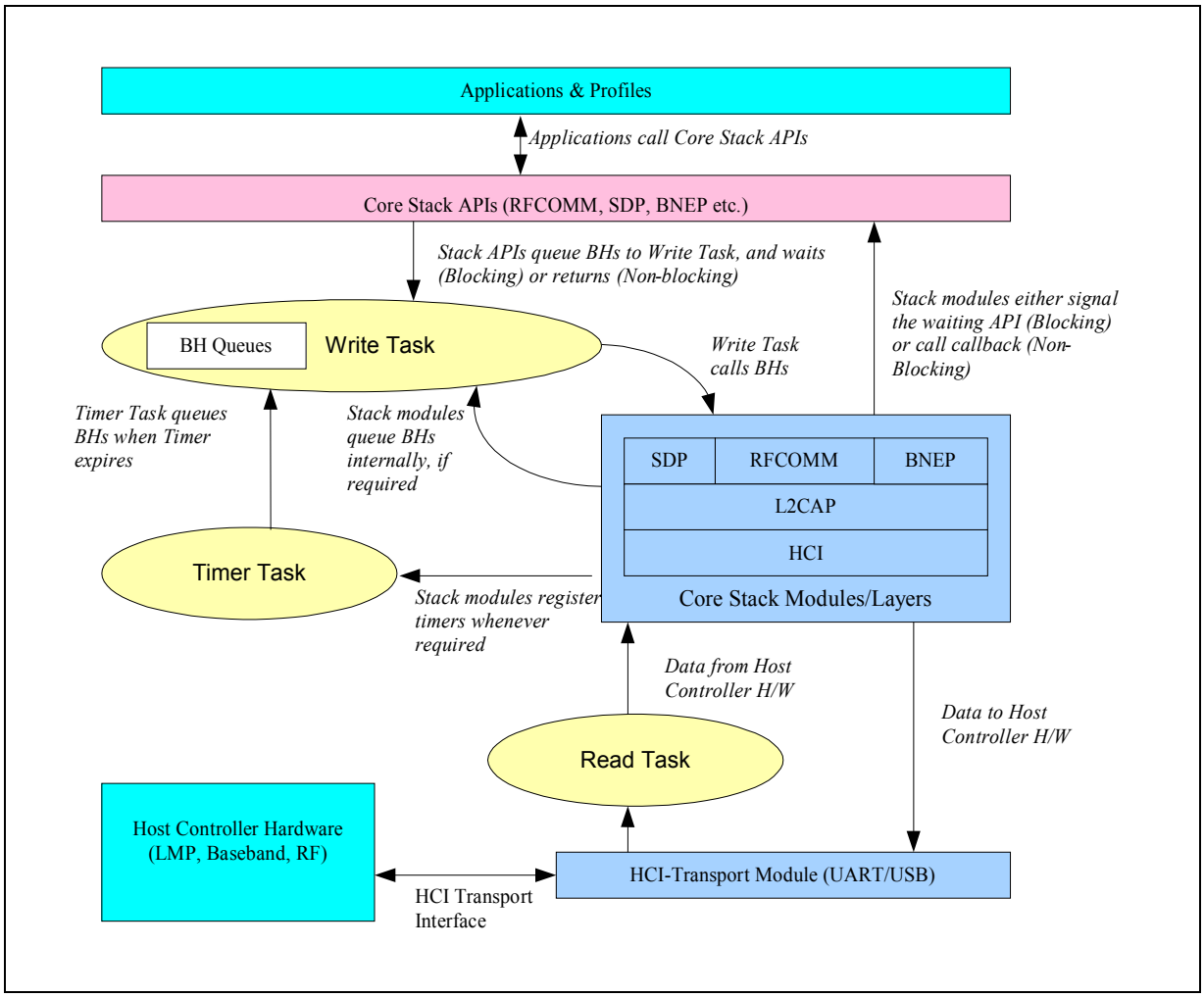
If the APIs are configured to be NON-BLOCKING, callback functions are required to be registered with the stack. The NON-BLOCKING API implies that the task that utilizes a service provided by the stack/profiles, does not block until the service is completed. The stack calls the callback functions asynchronously, on the completion of the function. Details on the callback functions and the registration process are described in the stack API documents [3].

If the APIs are configured to be BLOCKING, the APIs do not return till the entire function is completed. However, even in case of BLOCKING APIs, multiple tasks can make the function calls concurrently. The design of the internal stack is non-blocking to support simultaneous access, but portrays a blocking front-end to the application.

EtherMind is also designed to be platform and transport independent, by isolating and abstracting platform dependent portions sufficiently from core protocol/profile modules.

Task Architecture of EtherMind Stack

The following figure depicts the task architecture of EtherMind Host Protocol Stack system.



As evident from the figure above, the entire stack executes in the context of two main tasks - the Write Task and the Read Task.

Read, Write & Timer Tasks

The Stack APIs, when called by the applications, queue their respective Bottom Half Handlers (functions that perform the actual job of the APIs). The Write Task continuously services the Bottom Half (BH) Queues, and executes the queued Bottom Halves with the registered parameters one after another. The Write Task queues Bottom Halves based on a certain priority given to the individual queues (eg. timer expiration BHs and HCI data transmission BH are given preference over User API & Data BHs).

Applications & modules outside the stack can also use the Write Task to communicate with the stack when necessary – for example, it is particularly useful for implementing interrupt service routines (in cases where asynchronous events, such as timer events, are delivered to the stack in the form of interrupts).

The Read Task reads incoming HCI packets (ACL/SCO Data & Events) from the Host Controller Hardware into a circular buffer, waits for a complete HCI packet and then passes it to the HCI layer for further processing. The data is either delivered to the application or it is queued on write task for further action (e.g. HCI events). The circular read buffer should be filled with data read from the operating system specific Transport Layer's read interface mechanism. There are two mechanisms for filling this circular buffer:

- By registering a callback function with the read interface of the Transport Layer
- By having another task to continuously poll the read end-point and copy the data into the circular buffer.

The Timer Task shown in the diagram above is an optional task. The Timer Task is designed to provide all timer related functionalities of the stack. The timer library starts a task, which waits on timer events. When a timer event occurs, the timer task adds an BH function to the Write Task BH queue and goes back to wait for the next timer event. The implementation of the timer task is platform dependent.

The operating system, or, the platform may already provide a flexible timer implementation – in that case, the stack can use the operating system provided timer implementation, instead of the Timer Task.

Two-Task vs. Single-Task Model

EtherMind Stack can be configured to execute as a two-task model or a single-task model using a compilation time flag. Applications that require large amounts of data transfer (e.g. PAN) work better with the two-task architecture. However, for a two-task architecture, the operating system should support well-defined task scheduling mechanism.

In the single-task model, the entire stack and application work in the context of the same task. In this case, the Write Task performs the job of the Read Task, and the Timer Task if any.

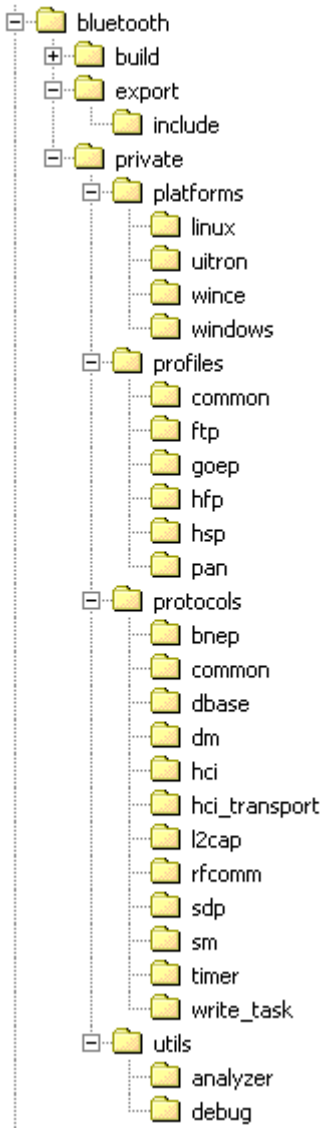
The task architecture, as described above, is specific to a platform. All generic platforms, like Windows, Linux, WinCE, are configured to use the Two Task Architecture. The Single Task Architecture is typically supported on the Embedded Stack Configuration.

Things to know before porting

Source Code Directory Structure

To be able to port the stack to a new operating system/platform, it is important to note how the source code is organized. The directory structure of the EtherMind Host Protocol Stack source repository is given in the figure below. It should be noted that the figure below is only an illustration; actual content may vary depending upon the release requirements.

The `bluetooth\export\` directory contains the stack header files, and application source codes, which are released along with the EtherMind SDK (Software development Kit). The `bluetooth\private\` folder contains the source code specific to the EtherMind Host Protocol Stack & Profiles.



Source code organization conventions are as follows:

- Each sub-directory (also referred to as source directory) represents an application, profile, protocol layer etc. E.g. `bnep, sdp, ftp etc.`
- All protocol layers and modules, e.g. `bnep`, are placed under the `bluetooth\private\protocols\` sub-directory.
- All profiles and related layers/modules are placed under the `bluetooth\private\profiles\` sub-directory.
- All stack (common, protocol and profile) header files, e.g. `BT_hci_api.h`, are placed under the `bluetooth\export\include\` sub-directory.
- All application, either protocol or profile, source codes are placed under the `bluetooth\export\` sub-directory.
- Build directories, for building stack protocol layers and modules and/or profiles, are placed at the top-level `bluetooth\build\` sub-directory. The build directory may further contain `lib` and `bin` directories to store the stack library/dll and the executable.
- Utility modules, such as `debug` and `analyzer`, are placed under the `bluetooth\private\utils\` sub-directory.
- The `bluetooth\private\platforms\` contains directories that contain files specific to platforms for which the protocol layers and profile are built. For example, this folder has the directories `linux, uitron, windows etc.` Whenever support must be added to new platforms, a directory is created inside this – and corresponding files, containing platform specific source codes, are added (e.g. driver files, HCI transport, write task, common platform initialization & shutdown routines, and the OS Abstraction files. Refer to any of the existing directories).

The stack module routines, which needs to makes use of platform/operating system specific primitives, are isolated and placed under the `bluetooth\private\platforms\` sub-directory in file named with

<module>_pl.c. For example, platform-specific part of the Write Task is coded in the write_task_pl.c, available under the bluetooth\private\platforms\ sub-directory.

Operating System and Platform Interfaces

The application may use several features provided by the underlying Operating System like creation of tasks/threads, Memory management, Synchronization mechanisms, User Interface Mechanisms, access to peripherals etc. It is recommended that applications use the features and functionalities provided by the Operating System and/or Platform directly. EtherMind OS Abstraction Layer provides a very small sub-set of the functions required by an application. The OS Abstraction Layer is meant for easy porting of the EtherMind Stack.

The table below provides an overview of these platform-specific components of the EtherMind stack source codes.

Module	Files	Remarks
OS Abstraction Module	BT_os.h BT_os.c	The EtherMind OS Abstraction source code for a specific platform/operating system. The BT_os.h defines the basic data types, e.g. UINT32, that the stack modules (protocols & profiles) use in their source codes. The BT_os.c exports library functions for various platform specific operations, such as, creation of tasks/threads, task/thread synchronization, memory management etc.
Platform specific Initialization and Shutdown of EtherMind Stack modules	BT_common_pl.h BT_common_pl.c	These files contain the platform specific initialization and shutdown routines for the EtherMind stack.
Serial Port Configuration	BT_serial.h BT_serial.c	These files contain generic routine to configure serial port for a platform/operating system – for example, baud rate, hardware flow/control, stop bits etc.
HCI UART Transport Module	uart.h hci_uart_internal.h hci_uart.c	Platform specific routines for the UART Transport module. This module is responsible for reading the COM/Serial port (to which the Bluetooth H/W is connected) in the platform-specific manner and delivers the data read to the HCI Transport module. Also, it provides APIs to write data to the COM/Serial port.
HCI USB Transport Module	usb.h hci_usb_internal.h hci_usb.c	Platform specific routines for the USB Transport module. This module is responsible for reading the USB port/device (to which the Bluetooth H/W is connected) in the platform-specific manner and delivers the data read to the HCI Transport module. Also, it provides APIs to write data to the USB port/device.
The Read Task	read_task_pl.c	Platform-specific component of the Read Task module – primarily for the creation of the Read Task and synchronization.
The Write Task	write_task_pl.c	Platform-specific component of the Write Task module – primarily for the creation of the Write Task and synchronization.

With exception of the OS Abstraction module, all of the above may not need to be ported on a new operating system/platform. A thorough study of the new operating system and an evaluation of the application usage model are required before deciding the list of the modules required to be ported. For example, the File Operations related files are not required to be ported if profiles that need it are not required on the new platform/operating system.

Porting need of other EtherMind modules

Even though EtherMind stack is designed to use only OS Abstraction module and modules from the platforms directory, there are few EtherMind modules that may need to undergo certain changes, to be used suitably under the new platform/operating system. These are outlined in the table below.

Module	Files	Remarks
EtherMind Timer Library	BT_timer.c	The implementation of EtherMind Timer Library creates the Timer Task at the time of its initialization, using the primitives available from the OS Abstraction module. As stated earlier, the platform may provide its timer functionalities, and in that case having a separate Timer Task may be costly. Even when a Timer Task is used, as available in the implementation of EtherMind Timer Library, specific support required from the operating system to obtain the current time – since, this is platform-specific, porting may be required.
EtherMind Debug Library	BT_debug_internal.h BT_debug.c	The default implementation of EtherMind Debug Library opens a log file named as "debug.log" and the debug logs from the stack modules are output in the log file. It may be required that the debug library behaves differently, or, opens a different log file, or, outputs the logs to a serial port instead of a log file. Hence, there may be a need to port the Debug Library for the new platform.

Porting EtherMind OS Abstraction Module

The EtherMind OS Abstraction module, consisting of BT_os.h and BT_os.c, are typically placed under the platforms sub-directory for a specific operating system/platform.

Porting OS Abstraction Data Types

The platform specific data types are abstracted and mapped to data types used within the EtherMind Stack, and are defined in the Header file BT_os.h. These data types may be used by the applications to make the applications platform independent. In addition, the BT_os.h also contains function declarations of OS Abstraction primitives.

Detailed description and porting guide for the data types exported by the BT_os.h is given in the table below.

Data Type	Remarks/Porting Guide
Basic Data Types	
INT8	Defined to be equivalent to 'char' of size 1 octet
CHAR	Defined to be equivalent to 'char' of size 1 octet
UINT8	Defined to be equivalent to 'unsigned char' of size 1 octet
UCHAR	Defined to be equivalent to 'unsigned char' of size 1 octet
INT16	Defined to be equivalent to 'short int' of size 2 octet
UINT16	Defined to be equivalent to 'unsigned short int' of size 2 octet
INT32	Defined to be equivalent to 'long int' of size 4 octet
UINT32	Defined to be equivalent to 'unsigned long int' of size 4 octet
BOOLEAN	Defined to be equivalent to 'unsigned char' of size 1 octet
UINT64	Defined to be equivalent to 'double' of size 8 octet
Task/Thread Related Data Types	
BT_thread_type	This data type is defined to be equivalent to the Thread or Task Identifier for the platform or operating system on which the OS Abstraction is being ported. Example: • On Windows (User Mode) platforms, EtherMind OS Abstraction uses the Windows Threads functions for thread creation and synchronization. And, hence, the BT_thread_type is defined to be HANDLE. • On Linux (User Mode) platforms, EtherMind OS Abstraction uses the POSIX Thread library for thread creation and synchronization. Hence, the BT_thread_type is defined to be pthread_t.
BT_thread_attr_type	This data type is defined to be equivalent of thread/task Attributes, and used at the time of creation of the thread/task. This may be defined as a structure containing the thread/task attributes (such as, priority, scheduling algorithm, stack size etc.), as needed by the thread/task model of the underlying operating system. Example: • On Windows (User Mode) platforms, EtherMind OS Abstraction uses the Windows Threads functions for thread creation and synchronization. And, hence, the BT_thread_attr_type is defined to be SECURITY_ATTRIBUTES. • On Linux (User Mode) platforms, EtherMind OS Abstraction uses the POSIX Thread library for thread creation and synchronization. Hence, the BT_thread_attr_type is defined to be pthread_attr_t.

Data Type	Remarks/Porting Guide
BT_thread_mutex_type	This data type is defined to represent a Mutex object. A mutex is a Mutual Exclusion device, and is useful for protecting shared data structures from concurrent modifications, and implementing critical sections and monitors. Every EtherMind stack module protects its global data using its own mutex object by declaring a variable of BT_thread_mutex_type data type, and initializing it. Also, it is expected that the declared mutex acts like a Binary Semaphore, having the following properties: • A mutex has two possible states: unlocked (not owned by any task/thread), and locked (owned by one task/thread). • Two different tasks/threads can never own a mutex simultaneously. A task/thread attempting to lock a mutex that is already locked by another task/thread is suspended until the owning task/thread unlocks the mutex first. As a result, the OS Abstraction implementation should enforce the above rules/properties. Example: • On Windows (User Mode) platforms, EtherMind OS Abstraction uses the Windows Threads functions for thread creation and synchronization. And, hence, the BT_thread_mutex_type is defined to be HANDLE (Windows Thread Mutex Handle). • On Linux (User Mode) platforms, EtherMind OS Abstraction uses the POSIX Thread library for thread creation and synchronization. Hence, the BT_thread_mutex_type is defined to be pthread_mutex_t.
BT_thread_mutex_attr_type	This data type is defined to represent attributes of a Mutex object, and may be defined as a structure containing several mutex attributes (such as, fast/recursive/error-checking mutex, initial owner, named mutex etc.), as needed by the thread/task synchronization model of the underlying operating system. Example: • On Windows (User Mode) platforms, EtherMind OS Abstraction uses the Windows Threads functions for thread creation and synchronization. And, hence, the BT_thread_mutex_attr_type is defined to be SECURITY_ATTRIBUTES. • On Linux (User Mode) platforms, EtherMind OS Abstraction uses the POSIX Thread library for thread creation and synchronization. Hence, the BT_thread_mutex_attr_type is defined to be pthread_mutexattr_t.

Data Type	Remarks/Porting Guide
BT_thread_cond_type	This data type is defined to represent a Conditional Variable object. A Condition (short for Condition Variable) is a synchronization object that allows threads to suspend execution and relinquish the processors until some predicate on the shared data is satisfied. The basic operations on conditions are: • A task/thread waits on the Condition, suspending the task/thread execution until another task/thread signals the condition • A task/thread signals the condition (when the predicate becomes true) A condition variable must always be associated with a mutex object, to avoid the race condition where a thread prepares to wait on a condition variable and another thread signals the condition just before the first thread actually waits on it. Example: • On Windows (User Mode) platforms, EtherMind OS Abstraction uses the Windows Threads functions for thread creation and synchronization. And, hence, the BT_thread_cond_type is defined to be HANDLE (Windows Thread Event Handle). • On Linux (User Mode) platforms, EtherMind OS Abstraction uses the POSIX Thread library for thread creation and synchronization. Hence, the BT_thread_cond_type is defined to be pthread_cond_t.
BT_thread_cond_attr_type	This data type is defined to represent attributes of a Conditional Variable object, and may be defined as a structure containing several conditional variable attributes (such as, manual reset, initial state, named object etc.), as needed by the thread/task synchronization model of the underlying operating system. Example: • On Windows (User Mode) platforms, EtherMind OS Abstraction uses the Windows Threads functions for thread creation and synchronization. And, hence, the BT_thread_cond_attr_type is defined to be SECURITY_ATTRIBUTES. • On Linux (User Mode) platforms, EtherMind OS Abstraction uses the POSIX Thread library for thread creation and synchronization. Hence, the BT_thread_cond_attr_type is defined to be pthread_condattr_t.

Porting OS Abstraction Primitives

OS Abstraction Primitive	Purpose/Remarks
Task/Thread Creation Primitives	
BT_thread_create	To create a task/thread
BT_thread_attr_init	To initialize a task's/thread's attributes to default settings
BT_thread_detach	To put a task/thread to "detached" state, so that resources occupied by the task/thread is freed when the task/thread terminates
Task/Thread Synchronization Primitives	
BT_thread_mutex_init	To initialize a Mutex object
BT_thread_mutex_lock	To lock a Mutex object
BT_thread_mutex_unlock	To unlock a Mutex object
BT_thread_cond_init	To initialize a Conditional Variable object

OS Abstraction Primitive	Purpose/Remarks
BT_thread_cond_wait	To wait on a Conditional Variable object
BT_thread_cond_signal	To signal a Conditional Variable object
Memory Management Primitives	
BT_alloc_mem	To allocate memory dynamically
BT_free_mem	To free dynamically allocated memory
BT_mem_copy	To copy memory from one location to another
Task/Thread Delay Primitives	
BT_sleep	To delay execution for a specified number of seconds
BT_usleep	To delay execution for a specified number of microseconds

BT_thread_create

Synopsis

```
INT32 BT_thread_create
      (
          BT_thread_type         * thread,
          BT_thread_attr_type    * thread_attr,
          void * (* thread_start_routine) (void *),
          void                   * thread_args
      );
```

IN Parameters

`thread_attr`

Pointer to a caller allocated `BT_thread_attr_type` variable, pre-initialized using `BT_thread_attr_init()`, and appropriately populated by the caller.

`thread_start_routine`

This function pointer is the start-routine for the newly created task/thread. Upon creation, the new task/thread calls this function, passing it `thread_args` as the argument.

`thread_args`

As stated above, this parameter points to a caller allocated "resident" memory location, containing the arguments to be passed to the newly created task's/thread's start routine.

OUT Parameters

`thread`

Pointer to a caller allocated `BT_thread_type` variable.

Return Value

`INT32`: 0 on success, and -1 or error.

Porting Guide

On successful creation of the task/thread, the `thread` parameter should be populated with the newly created task's/thread's handle/identifier, as required by the task/thread management functions of the underlying operating system.

The caller may choose to set NULL for the `thread_attr` parameter. In that case, the

new task/thread should be created with default attribute values, as permitted and visualized by the task/thread functions of the underlying operating system.

The task/thread start routine, `thread_start_routine`, returns a "`void *`" data type, typically, pointer to a memory location containing the task's/thread's exit status. When this start routine returns from its execution, the created task/thread should be terminated as well.

Under certain usage scenarios, it may not be advisable to create threads/tasks dynamically on certain operating system/platform. Instead, a task/thread pool may be available. In this case, `BT_thread_create()` should attempt to attach to a (predefined) task/thread from the pool.

It should also be noted that no restriction is imposed on the user of `BT_thread_create()` that the thread parameter would be resident (global) in memory. Hence, it will be wrong for `BT_thread_create()` to assume so in its implementation.

BT_thread_attr_init

Synopsis

```
INT32 BT_thread_attr_init
      (
          BT_thread_attr_type    * thread_attr
      );
```

IN Parameters None.

OUT Parameters `thread_attr`

Pointer to a caller allocated `BT_thread_attr_type` variable to be initialized to default task/thread attribute values.

Return Value `INT32`: 0 on success, and -1 or error.

Porting Guide This primitive is required to initialize the `thread_attr` with default task/thread attribute values, as permitted and visualized by the task/thread functions of the underlying operating system.

BT_thread_detach

Synopsis

```
INT32 BT_thread_detach ( void );
```

IN Parameters None.

OUT Parameters None.

Return Value `INT32`: 0 on success, and -1 or error.

Porting Guide This primitive is required to put the calling task/thread into "detached" state. The "detached" state guarantees that the memory resources consumed by the task/thread will be freed immediately when the task/thread terminates (equivalent to returning from the task's/thread's start routine). This also prevents other tasks/threads from synchronizing on the termination of the task/thread in "detached" state (by using task/thread "join" functionality, if provided, by the task/thread function of the underlying operating system.

The `BT_thread_detach` does not take a parameter. However, it is logical that the thread identifier/handle (`BT_thread_type`) needs to be known to put the task/thread into "detached" state, the same should be obtained using operating system/platform specific manner in the implementation of this primitive [for example, on Linux (User Mode) platform, `pthread_self()` is used to obtain the calling thread's identifier].

It should be noted that EtherMind Tasks (the Read, Write and the optional Timer tasks) are designed such that they never terminate. These tasks are designed to go into "sleep" state when the task is idle, and "wakes up" whenever required. Hence, this primitive is not required unless EtherMind tasks are re-designed to terminate.

However, this OS Abstraction primitive should be implemented when applications are written to use the OS Abstraction module for task/thread management, and they may terminate and the resources allocated to be freed.

BT_thread_mutex_init

Synopsis

```
INT32 BT_thread_mutex_init
      (
          BT_thread_mutex_type         * mutex,
          BT_thread_mutex_attr_type    * mutex_attr
      );
```

IN Parameters `mutex_attr`

Pointer to a caller allocated `BT_thread_mutex_attr_type` variable to be containing the attribute values with which the mutex object to be initialized.

OUT Parameters `mutex`

Pointer to a caller allocated `BT_thread_mutex_type` variable to be initialized, to attribute values as given with `mutex_attr`.

Return Value `INT32`: 0 on success, and -1 or error.

Porting Guide This primitive is required to initialize the specified mutex object with the specified mutex attributes.

It is expected that the mutex object acts like a Binary Semaphore, having the following properties:

- A mutex has two possible states: unlocked (not owned by any task/thread),

and locked (owned by one task/thread).

- Two different tasks/threads can never own a mutex simultaneously. A task/thread attempting to lock a mutex that is already locked by another task/thread is suspended until the owning task/thread unlocks the mutex first.

As a result, implementation of this primitive should enforce the above rules/properties.

The caller may choose to set NULL for the `mutex_attr` parameter. In that case, the mutex object should be initialized with default attribute values, as permitted and visualized by the task/thread functions of the underlying operating system.

BT_thread_mutex_lock

Synopsis

```
INT32 BT_thread_mutex_lock
      (
          BT_thread_mutex_type    * mutex
      );
```

IN Parameters None.

OUT Parameters

`mutex`

Pointer to a caller allocated `BT_thread_mutex_type` variable to be initialized, which needs to be locked.

Return Value `INT32`: 0 on success, and -1 or error.

Porting Guide This primitive attempts to lock the specified mutex object.

If the specified mutex object is in the "locked" state, the implementation of this primitive should suspend execution of the task/thread, and relinquish the processor until the mutex object becomes available in "unlocked" state.

It is advisable to implement this primitive to test for "deadlock" situation when it is called by the same task/thread who currently owns the mutex object. If this "deadlock" is asserted, error (-1) must be returned.

BT_thread_mutex_unlock

Synopsis

```
INT32 BT_thread_mutex_lock
      (
          BT_thread_mutex_type    * mutex
      );
```

IN None.

Parameters

OUT Parameters `mutex`

Pointer to a caller allocated `BT_thread_mutex_type` variable to be initialized, which needs to be unlocked.

Return Value `INT32`: 0 on success, and -1 or error.

Porting Guide This primitive attempts to unlock the specified mutex object.

BT_thread_cond_init

Synopsis

```
INT32 BT_thread_cond_init
      (
          BT_thread_cond_type         * cond,
          BT_thread_cond_attr_type    * cond_attr
      );
```

IN Parameters `cond_attr`

Pointer to a caller allocated `BT_thread_cond_attr_type` variable to be containing the attribute values with which the conditional variable object to be initialized.

OUT Parameters `cond`

Pointer to a caller allocated `BT_thread_cond_type` variable to be initialized, to attribute values as given with `cond_attr`.

Return Value `INT32`: 0 on success, and -1 or error.

Porting Guide This primitive initializes a specified conditional variable object with the specified attributes.

A Condition (short for Condition Variable) is a synchronization object that allows threads to suspend execution and relinquish the processors until some predicate on the shared data is satisfied. The basic operations on conditions are:

- A task/thread waits on the Condition, suspending the task/thread execution until another task/thread signals the condition
- A task/thread signals the condition (when the predicate becomes true)

A condition variable must always be associated with a mutex object, to avoid the race condition where a thread prepares to wait on a condition variable and another thread signals the condition just before the first thread actually waits on it.

The caller may choose to set NULL for the `cond_attr` parameter. In that case, the conditional variable should be initialized with default attribute values, as permitted and visualized by the task/thread functions of the underlying operating system.

BT_thread_cond_wait

Synopsis

```
INT32 BT_thread_cond_wait
      (
          BT_thread_cond_type    * cond,
          BT_thread_mutex_type   * cond_mutex,
      );
```

IN Parameters

None.

OUT Parameters

`cond`

Pointer to a caller allocated `BT_thread_cond_type` variable on which the caller task/thread needs to wait.

`cond_mutex`

Pointer to a caller allocated `BT_thread_mutex_type` variable which will be used to synchronize actions on the conditional variable `cond`.

Return Value

`INT32`: 0 on success, and -1 or error.

Porting Guide

This primitive implements waiting on a conditional variable.

The `cond_mutex` parameter represents the caller allocated mutex object which is associated with the conditional variable `cond`, to avoid the race condition where the caller thread prepares to wait on the condition variable and another thread signals the condition just before the caller thread actually waits on it.

The implementation of `BT_thread_cond_wait()` unlocks the `cond_mutex` (as per `BT_thread_mutex_lock()`), and waits for the conditional variable `cond` to be signaled. The task/thread execution is suspended and does not consume CPU time until the condition `cond` is signaled. The mutex object `cond_mutex` must be locked by the calling task/thread on entrance to `BT_thread_cond_wait()`. Before returning to the calling task/thread, `BT_thread_cond_wait()` reacquires the `cond_mutex` (as per `BT_thread_mutex_lock()`).

Unlocking the `cond_mutex` and subsequent suspending execution on the conditional variable `cond` should be performed “atomically”. Thus, if all tasks/threads always acquire the `cond_mutex`, before signalling the condition `cond`, this guarantees that the condition cannot be signaled (and be ignored) between the time a task/thread locks the `cond_mutex` and the time it waits on the condition `cond`.

BT_thread_cond_signal

Synopsis

```
INT32 BT_thread_cond_signal
      (
          BT_thread_cond_type    * cond
```

```
);
```

IN Parameters None.

OUT Parameters `cond`

Pointer to a caller allocated `BT_thread_cond_type` variable that the caller task/thread needs to signal.

Return Value `INT32`: 0 on success, and -1 or error.

Porting Guide This primitive implement signalling a conditional variable.

Implementation of this primitive restarts one of the tasks/threads currently waiting on the conditional variable `cond`. If no task/thread is waiting on `cond`, nothing should happen and no error should be generated, that is, the signal will be assumed to be lost. If more than one task/thread are waiting on the `cond`, then exactly one of them should be awakened and restarted for execution, but there is no rule specified as to which one.

BT_alloc_mem

Synopsis

```
void * BT_alloc_mem
       (
           UINT32   nbytes
       );
```

IN Parameters `nbytes`

Size of the memory block to be dynamically allocated.

OUT Parameters None.

Return Value `void *`: Pointer to the allocated memory segment, or, NULL in case of error.

Porting Guide This primitive is required to implement dynamic memory allocation.

The implementation may make use of available library calls (such as, ANSI `malloc()`) to dynamically allocate the requested size of memory, or, it may implementation allocation from a fixed size memory pools/blocks. The choice of implementation is specific to the platform/operating system in question, design scenario and availability of memory.

The only and most important requirement on this primitive is that the allocated memory should be “resident” (global), addressable and readable/writeable by all tasks/threads of the EtherMind stack and application system.

BT_free_mem

Synopsis

```
void BT_free_mem
```

```
(
    void    * ptr
);
```

IN Parameters

`ptr`

Pointer to the dynamically allocated memory segment that needs to be freed.

OUT Parameters

None.

Return Value

None.

Porting Guide

This primitive is required to implement freeing of dynamically allocated memory segment.

The implementation may make use of available library calls (such as, ANSI `free()`), if available, to free the dynamically allocated memory segment, or, it may implement freeing a fixed size memory block to a globally available memory pools/blocks. The choice of implementation is specific to the platform/operating system in question, design scenario and availability of memory.

The only and most important requirement on this primitive is that the freeing mechanism must be consistent with implementation of `BT_alloc_mem()`.

BT_mem_copy

Synopsis

```
void * BT_mem_copy
       (
           void     * dest,
           void     * src,
           UINT32   size,
       );
```

IN Parameters

`dest`

Destination memory location to where content of the `src` to be copied

`src`

Source memory location from where content needs to be copied.

`size`

Size of the memory block to be copied.

OUT Parameters

None.

Return Value

`void *`: Pointer to the destination memory location.

Porting Guide This primitive implements copying content of memory location `src`, to destination location `dest`, of `size` number of octets.

BT_sleep

Synopsis

```
void BT_sleep
     (
         UINT32   num_seconds
     );
```

IN Parameters `num_seconds`

Number of seconds that a task/thread needs to sleep/delay

OUT Parameters None.

Return Value None.

Porting Guide This primitive implements delay of execution or sleeping for a given number of seconds.

The implementation of this primitive must ensure that the calling task/thread relinquishes control of the CPU for the entire duration of sleep, as specified by number of seconds.

BT_usleep

Synopsis

```
void BT_usleep
     (
         UINT32   num_micro_seconds
     );
```

IN Parameters `num_micro_seconds`

Number of microseconds that a task/thread needs to sleep/delay

OUT Parameters None.

Return Value None.

Porting Guide This primitive implements delay of execution or sleeping for a given number of microseconds.

The implementation of this primitive must ensure that the calling task/thread relinquishes control of the CPU for the entire duration of sleep, as specified by the number of microseconds.

Customizing Module Initialization & Shutdown

This section summarizes the requirements of stack configuration for a platform. This is not really "porting", rather configuration of the stack modules depending on the usage needs of the platform concerned.

For example, configuring the BNEP module for EtherMind stack running in Windows user mode does not help, as the PAN profile, which uses BNEP for its functionalities, needs to be in the kernel mode. In this case, Windows user mode builds may exclude BNEP, and corresponding initialization and shutdown need not be done.

The EtherMind Stack provides three fundamental initialization & shutdown routines:

- BT_ethermind_init

  This API initializes the EtherMind Stack, and its modules. Primary objective of this routine is to create the EtherMind tasks (the Read, Write, and optional Timer tasks), initialization of Mutex and Conditional Variable objects of various modules, protocols and profiles.

- BT_bluetooth_on

  This API switches on the Bluetooth functionality of the EtherMind Stack. As a part of this initialization, the stack resets the Bluetooth hardware, retrieves Bluetooth Device Address, and performs other initialization required to get the modules ready of Bluetooth operations.

- BT_bluetooth_off

  This API performs shutdown of the Bluetooth functionality of the EtherMind Stack.

For more details on the above APIs, refer to EtherMind Stack API Document [3].

The application is required to call the `BT_ethermind_init()` only once, for the lifetime of the stack, and prior to any other stack APIs. However, `BT_bluetooth_on()` & `BT_bluetooth_off()` can be called one after the other, in cyclic fashion. This provides flexibility to the user application that it can switch on Bluetooth functionalities only when it is required, and can switch it off at other times to save precious resources (computational power, memory, battery power etc.).

Actual work done by each of the above APIs is sub-divided into three parts:

- Initialization/shutdown of the Core Stack modules. The Core Stack modules are always needs to be present, in every configuration of the stack, and they provide basic Stack and Bluetooth functionalities.

  The Core Stack modules are, in the order in which they are initialized: Debug Library, Write Task, Timer Library, HCI Transport, HCI and L2CAP.

- Initialization/shutdown of the modules that appear below the Core Stack modules. Initialization (and, shutdown) of these modules depends on the need and configuration.

  Each of the above APIs call a sub-routine, named with "`_lower_pl`" in the end, to perform initialization/shutdown of these modules. In case of `BT_ethermind_init()` and `BT_bluetooth_on()`, the "`_lower_pl`" routines are called after HCI Transport module, and

before HCI, is initialized. On the other hand, in case of BT_bluetooth_off(), the "_lower_pl" routines are called after HCI, and before HCI Transport, is shutdown.

Examples of such modules: HCI UART or USB or BCSP Transport

- Initialization/shutdown of the modules that appear above the Core Stack modules. Initialization (and, shutdown) of these modules depends on the need and configuration.

  Each of the above APIs call a sub-routine, named with "_upper_pl" in the end, to perform initialization/shutdown of these modules. In case of BT_ethermind_init() and BT_bluetooth_on(), the "_upper_pl" routines are called after all the Core Stack modules are initialized. On the other hand, in case of BT_bluetooth_off(), the "_upper_pl" routines are called before Core Stack modules are shutdown.

  Examples of such modules: DM, SM, SDP, RFCOMM, TCS, BNEP, AVDTP etc.

The list of "_lower_pl" and "_upper_pl" routines, are as follows (all available in the BT_common_pl.c file under the private\platforms\ sub-directory for the platform):

- ethermind_init_lower_pl
- ethermind_init_upper_pl
- bluetooth_on_upper_pl
- bluetooth_on_lower_pl
- bluetooth_off_upper_pl
- bluetooth_off_lower_pl

The above routines may require to be configured, as required, for a platform. Configuration may also be done with the help of module-inclusion compilation flags.

Porting Serial Port Configuration Module

The BT_serial.c & BT_serial.h files in the platforms/ sub-directory abstracts the configuration of Serial/COM Port under different operating systems and platforms, particularly for use in the HCI UART Transport Layer module. These files may required to be ported for the new operating system, if HCI UART Transport Layer is planned to be used.

The BT_serial.c exports an API 'BT_set_serial_params()' which can be used to set/configure serial/COM port (UART) parameters, using a data type 'BT_SERIAL_PARAMS', defined in BT_serial.h.

Description of parameters of BT_SERIAL_PARAMS is given in the table below.

Parameter	Data Type	Remarks
buadrate	int	The serial baud rate in bits per second

Parameter	Data Type	Remarks
use_hw_flow	unsigned char	Flag to indicate whether hardware flow control (CTS/RTS) needs to be used. • 0x00 – No hardware flow control • 0x01 – Hardware flow control to be configured, if possible
use_parity	unsigned char	Flag to indicate whether parity is required to be used • 0x00 – Parity generation & check to be disabled • 0x01 - Parity generation & check to be enabled
parity_type	unsigned char	Type of parity to use • 0x00 – Even parity • 0x01 – Odd parity
parity_char	unsigned char	Parity error replacement character

The BT_serial.h also defines a data type called 'BT_SERIAL_FD' to represent the handle/descriptor for the opened serial device. While porting, this data type needs to be defined to appropriate data type to represent an opened serial device descriptor/handle for the intended platform/operating system.

The caller of the BT_set_serial_params() API fills up a variable of type BT_SERIAL_PARAMS, and passed it to the API. The implementation of BT_set_serial_params() may require changes for a platform or operating system depending upon the kind of configuration interface provided by the serial driver; the porting guide for the same is given below.

Synopsis

```
int BT_set_serial_params
    (
        BT_SERIAL_FD        serial_fd,
        BT_SERIAL_PARAMS    * serial_params
    );
```

IN Parameters

serial_fd

This parameter represents the descriptor/handle of the opened serial device for which given configuration parameters serial_params needs to be set.

serial_params

Pointer to the caller allocated BT_SERIAL_PARAMS variable containing the serial device configuration parameters to be set.

OUT Parameters

None.

Return Value

int: 0 on success, and -1 or error.

Porting Guide

This API implements the procedure required to set the given configuration parameters serial_params to the serial device as represented by serial_fd.

It should be noted that the HCI UART Transport module primarily uses this API. As per the Bluetooth specification, number of data bits to be configured for UART is 8, and number of stop bit is 1. These settings are automatically done by this API, and should be hard-coded in its implementation.

Example:

- On Windows, the `SetCommState()` and `SetCommTimeouts()` APIs are used to implement serial port configuration. The BT_SERIAL_FD is defined to be of data type `HANDLE`.

Porting HCI UART Transport Module

The HCI UART Transport module of the EtherMind stack consists of the following key routines, which are available in the `hci_uart.c` (or, `win_uart.c` on Windows) file:

HCI-UART Transport Routines	Purpose/Remarks
`uart_set_serial_settings`	To set serial device (COM Port) to open and the baud rate to configure during initialization of the HCI-UART module
`uart_init`	The `BT_ethermind_init()` handler for the HCI-UART Transport module
`uart_bt_init`	The `BT_bluetooth_on()` handler for the HCI-UART Transport module
`uart_bt_shutdown`	The `BT_bluetooth_off()` handler for the HCI-UART Transport module
`uart_line_read`	To read data from the UART or the serial device
`uart_send_data`	API exported for HCI to send data to the UART
`uart_write_data`	To write data to the UART or the serial device

Depending on the platform/operating system, these routines need to be implemented to perform actual job of reading/writing data to/from the stack (HCI).

Porting HCI-UART Transport Routines

uart_set_serial_params

Synopsis

```
void uart_set_serial_params
     (
         CHAR     * serial_device,
         UINT32   baudrate
     );
```

IN Parameters

`serial_device`

This parameter is a caller allocated character array, which holds the name of the serial device to open. For example, “COM1” on Windows, or, “/dev/ttyS0” on Linux.

`baudrate`

The serial baud rate to configure for the HCI-UART interface.

OUT Parameters None.

Return Value None.

Porting Guide This API is called by the application even before `BT_ethermind_init()` is called, if the intended HCI interface is HCI-UART Transport layer, to configure the serial port to open and the baud rate to set for the same.

The actual opening of the device, and its configuration, is performed much later when `uart_bt_init()` gets called from the context of `BT_bluetooth_on()`. Hence this API is only required to store the information provided by the application in a global variable and use it later when required.

uart_init

Synopsis

```
void uart_init ( void );
```

IN Parameters None.

OUT Parameters None.

Return Value None.

Porting Guide This is the `BT_ethermind_init()` handler for the HCI-UART Transport layer, and called from the context of the same if `BT_UART` compilation flag is used during compilation.

This routine is expected to initialize the HCI-UART Transport layer specific Mutex object (using `BT_thread_mutex_init()`).

Additionally, this routine may also create a task/thread to perform the job of reading data from the serial device driver of the underlying platform/operating system. It should be noted that the creation of this task is not necessary for all platforms – this task is not required on platforms where the serial device itself delivers data asynchronously to the HCI-UART Transport Layer by some mechanism (e.g., via interrupts).

If a task/thread is created, it must be put on wait (using `BT_thread_cond_wait()`), and should not attempt to reference the device since it is opened much later in the context of `uart_bt_init()`.

uart_bt_init

Synopsis

```
void uart_bt_init ( void );
```

IN Parameters None.

OUT Parameters None.

Return Value None.

Porting Guide This is the BT_bluetooth_on() initialization handler for the HCI-UART Transport layer, and called from the context of the same if BT_UART compilation flag is used during compilation.

This routine is expected to open the serial device and configure the same, in platform specific manner, as per the information provided with uart_set_serial_settings(). The configuration of the serial device may be done using BT_set_serial_params() API.

Once the serial device is opened and configured, and if a UART read task/thread was created during uart_init() to read data from UART device driver, it must signal the task/thread to wake up from sleep and start reading data from the serial device driver.

uart_bt_shutdown

Synopsis

```
void uart_bt_shutdown ( void );
```

IN Parameters None.

OUT Parameters None.

Return Value None.

Porting Guide This is the BT_bluetooth_off() shutdown handler for the HCI-UART Transport layer, and called from the context of the same if BT_UART compilation flag is used during compilation.

This routine is expected to close the serial device in platform specific manner.

Prior to closing the serial device, if a UART read task/thread was created during uart_init() to read data from UART device driver, it must signal the task/thread to go to sleep and stop it from referencing the serial device any further.

uart_line_read

Synopsis

```
void uart_line_read ( void );
```

IN Parameters None.

OUT Parameters None.

Return Value None.

Porting Guide The job of this routine is to retrieve incoming data from the UART serial device driver of the underlying platform/operating system and deliver the same to the HCI Transport module (the Read Task of EtherMind Stack) using the stack exported `hci_transport_enqueue()` API (declared in `hci_transport.h`).

If a UART read task/thread was created during `uart_init()` to read data from UART device driver, this routine may be called from the task's/thread's start routine to retrieve the incoming data and deliver to the stack.

The actual mechanism for retrieving data from the underlying serial device/driver is platform specific, and needs to be implemented in a manner most suitable for that platform.

uart_send_data

Synopsis

```
void uart_send_data
     (
         UCHAR    type,
         UCHAR    * buffer,
         UINT16   buffer_size,
         UCHAR    flag
     );
```

IN Parameters

`type`

This parameter defines the type of HCI packet being currently written to the UART serial device (and ultimately, to the Bluetooth hardware). The values can be, as defined in `hci_h.h` (HCI module):

- `HCI_COMMAND_PACKET`
- `HCI_ACL_DATA_PACKET`
- `HCI_SCO_DATA_PACKET`

As required by the HCI-UART Transport Specification, this packet `type` needs to precede the actual HCI packet.

`buffer`

HCI allocated buffer containing the HCI packet (segment) to be written to the UART serial device for transmission.

`buffer_size`

Size of the HCI allocated buffer containing the HCI packet (segment) to be written to the UART serial device for transmission.

`flag`

Since the `type` value needs to precede the actual HCI packet, this parameter states when to write the `type` to UART serial device.

OUT Parameters None.

Return Value None.

Porting Guide This is an API exported to the EtherMind stack, typically the HCI module, for writing HCI packets to the Bluetooth hardware connected via UART.

HCI may divide entire content of its packet into several segments and call this API several times to send the entire packet. However, the `type` of HCI packet needs to precede the entire packet, and must be written only before the first segment is written. This is controlled by the use of `flag` parameter. The `flag` parameter needs to be interpreted as follows:

- `flag` is `0x01` – write the byte as given by `type`, followed by `buffer` of `buffer_size`
- `flag` is `0x00` – write the `buffer` of `buffer_size` only

Implementation may call the `uart_write_data()` routine to perform actual job of transferring the data to be written onto the UART device driver for transmission in the platform-specific manner.

uart_write_data

Synopsis

```
void uart_write_data
     (
         UCHAR    * buffer,
         UINT16   buffer_size
     );
```

IN Parameters

`buffer`

Caller allocated buffer containing the HCI packet (segment) to be written to the UART serial device for transmission.

`buffer_size`

Size of the caller allocated buffer containing the HCI packet (segment) to be written to the UART serial device for transmission.

OUT Parameters None.

Return Value None.

Porting Guide The job of this routine is to transfer outgoing data to the UART serial device driver of the underlying platform/operating system for transmission to the Bluetooth hardware.

The actual mechanism for retrieving data from the underlying serial device/driver is platform specific, and needs to be implemented in a manner most suitable for that platform.

It should be noted that this routine is blocking in nature, that is, it should return only after writing the entire data that is required to be transmitted.

Porting HCI USB Transport Module

The HCI USB Transport module of the EtherMind stack consists of the following key routines, which are available in the `hci_usb.c` (or, `win_usb.c` on Windows) file:

HCI-UART Transport Routines	Purpose/Remarks
`usb_init`	The `BT_ethermind_init()` handler for the HCI-USB Transport module
`usb_bt_init`	The `BT_Bluetooth_on()` handler for the HCI-USB Transport module
`usb_bt_shutdown`	The `BT_Bluetooth_off()` handler for the HCI-USB Transport module
`usb_line_read`	To read data from the USB device driver
`usb_send_data`	API exported for HCI to send data to the USB
`usb_write_data`	To write data to the USB device driver

Depending on the platform/operating system, these routines need to be implemented to perform actual job of reading/writing data to/from the stack (HCI).

The implementation of HCI USB Transport module follows similar model as that of HCI UART Transport module. It is not described here again for the sake of repetition.

Porting EtherMind Read Task

The platform specific extensions for the EtherMind Read Task (part of HCI Transport module) are implemented in the `ht_read_task_pl.c`, available under the `bluetooth\private\platform\` sub-directory for the platform.

The platform specific routines of the Read Task, and their porting need, is described in the table below.

Read Task Platform Routines	Purpose/Remarks
ht_read_task_create_pl	This routine is called from the context of hci_transport_init(), the BT_ethermind_init() handler for the HCI-Transport module, and used for creating the Read Task/Thread in the platform specific manner. The implementation may create the task using BT_thread_create() primitive from the OS Abstraction module. One reason for keeping the task/thread creation platform-specific, and separate from the source code of HCI-Transport (private\protocols\hci_transport\) is that the task/thread may be created with platform/operating system specific parameters and configurations. These platform specific implementation extension (such as, setting priority or scheduling algorithm and stack size) can be implemented using the BT_thread_attr_type data type and BT_thread_attr_init() primitive prior to creation of the task. If the Read Task is not used, or, created by some other means, this routine may be left empty.
ht_read_task_shutdown_pl	This routine is called from the context of hci_transport_bt_shutdown(), the BT_bluetooth_off() handler for the HCI-Transport module, and used for cleaning up the dependencies on the Read Task, in the platform specific manner, when it is shutting down. It must be noted that EtherMind Tasks, including the Read Task, is not designed to terminate. It may be 'shutdown', which this means the task is put to sleep, and will be woken up when BT_bluetooth_on() gets called again. This routine needs to be implemented only when some clean-up required to be performed for a certain operating system/platform, when the task is put to sleep by the caller (hci_transport_bt_shutdown()) – such as, terminate a process waiting on this task and some signals are pending for it.

Porting EtherMind Write Task

The platform specific extensions for the EtherMind Write Task are implemented in the write_task_pl.c, available under the bluetooth\private\platform\ sub-directory for the platform.

The platform specific routines of the Write Task, and their porting need, is described in the table below.

Read Task Platform Routines	Purpose/Remarks
write_task_create_task_pl	This routine is called from the context of write_task_init(), the BT_ethermind_init() handler for the Write Task module, and used for creating the Write Task/Thread in the platform specific manner. The implementation may create the task using BT_thread_create() primitive from the OS Abstraction module. One reason for keeping the task/thread creation platform-specific, and separate from the source code of Write Task (private\protocols\write_task\) is that the task/thread may be created with platform/operating system specific parameters and configurations. These platform specific implementation extension (such as, setting priority or scheduling algorithm and stack size) can be implemented using the BT_thread_attr_type data type and BT_thread_attr_init() primitive prior to creation of the task. If the Write Task is not used, or, created by some other means, this routine may be left empty.

Porting EtherMind Timer Library

EtherMind stack exports a set of Timer APIs that is used by the EtherMind stack protocol layers for their timer needs. The EtherMind Timer Library is a software implementation and exports a set of APIs to start/stop and restart timers.

The default implementation of EtherMind Timer Library is built on top of the OS Abstraction module, and has its own task, the Timer Task, for tracking timeout events for registered timers. The minimum resolution of the timers registered in this implementation is 1 second. The resolution of the timer is dependent on the platform, and also on the frequency at which the Timer Task wakes up to check its queue of registered timers for timeout events.

The Timer Library APIs are designed with sufficient feature to enable platform specific implementation in the case where the underlying operating system provides timer functionalities. In that case, the timer task may not be required, however other Timer Library design components – such as, Timer Queues etc. – can be reused to provide timer functionality to the EtherMind stack modules and applications.

In this section, porting needs are described for the default implementation of the EtherMind Timer Library, assuming a Timer Task is created. For detailed description of the Timer Library APIs, refer to EtherMind Stack API Documentation [3].

Porting Needs of EtherMind Timer Library with Timer Task

All related source code for the EtherMind Timer Library is available in the BT_timer.c file under private\protocols\timer\ directory.

The Timer Task

The life cycle of the Timer Task is described below:

- The Timer Task is created in the timer_init() routine, which is the BT_ethermind_init() handler for the Timer module. Like other EtherMind tasks, this task is also put to sleep, by using BT_thread_cond_wait(), as soon as it is created, from its task/thread start-routine.

- The Timer Task is signaled to continue execution when `timer_bt_init()` is called, from the context of `BT_bluetooth_on()`. Thereafter, the Timer Task wakes up every 1 second from hibernation (implemented using `BT_sleep()`) and services its queue of active timer elements. If a timeout has happened for a timer, then it queues the timeout callback handler, as provided by the module at the time of starting the timer, to the Write Task, and goes on processing the next timer element.

  To decide whether timeout has occurred for an active timer, the Timer Task calls the `timer_get_current_time()` routine to retrieve the current system time, which it compares with the time at which the timer is required to be expired.

- When `timer_bt_shutdown()` is called, from the context of `BT_bluetooth_off()`, the Timer Task is again signaled to sleep, until `timer_bt_init()` is called again.

Since Timer Task implementation uses primitives only from the OS Abstraction module, porting requirements for this task is minimal, and would be similar to that of the Read Task and the Write Task. Hence, no other porting need is specified.

Getting Current System Time

The Timer Library calls the `timer_get_current_time()` routine to retrieve the current system time, which it used for the following purposes:

- To store the time at which a timer, currently being started by the user, needs to expire. It is arrived by adding the timeout value specified by the user to the current system time.

- To decide whether timeout has occurred for an active timer. It is decided by comparing the current system time with the expiry time value stored for the timer.

The mechanism to obtain the current system time is very much platform specific, and depends on the features provided by the underlying operating system. It should be noted that the "current system time" does not need to be time in absolute terms and meaning - for example, it can be "elapsed" time since the last system reset or boot-up.

The only requirement that the `timer_get_current_time()` must satisfy is as follows: it must return a positive integer (`UINT32`) value, which increments by the elapsed number of seconds in real-time when called again at a later point of time. For example, if `timer_get_current_time()` returns the value `1023786` now, then it must return `1023796` when called `10` seconds later.

For certain platforms/operating system (such as, Windows/Linux) which supports the `time()`, C run-time library call, the `timer_get_current_time()` can be implemented using `time()` since it returns number of seconds elapsed since the 'Epoch' [midnight (00:00:00), January 1, 1970, coordinated universal time (UTC)], according to the system clock.

Another approach for implementing `timer_get_current_time()` is to divide number of clock ticks since the last reset/boot-up by the clock ticks per second.

Porting Security Manager Persistent Storage

The EtherMind Security Manager (SM) provides a feature to store various SM configuration settings onto persistent storage media (such as, EEPROM or equivalent). This feature is available, and needs to be ported, only if Security Manager is compiled with the `SM_STORAGE` compilation flag.

When SM_STORAGE is defined during compilation, the Security Manager invokes the sm_storage_read() function during Bluetooth ON [BT_bluetooth_on()] procedure. The sm_storage_read() is responsible for reading various Security Manager configuration parameters from the persistent storage media.

Similarly, when SM_STORAGE is defined during compilation, the Security Manager invokes the sm_storage_write() function during Bluetooth OFF [BT_bluetooth_off()] procedure. The sm_storage_write() is responsible for writing various Security Manager configuration parameters to the persistent storage media.

Typically, the sm_storage_read() and sm_storage_write() functions are available in sm_storage.c file, under the bluetooth\private\platforms\ sub-directory for the platform/operating system, for which these have been ported.

The Security Manager configuration parameters, which needs to be read/written to/from the persistent storage, are described in the table below:

SM Configuration Parameter	Porting Guide
SM Security Mode	This parameter configures the Security Mode (1, 2 or 3) of the Security Manager. In sm_storage_write(), the current value of 'sm_security_mode' is written in the persistent storage. In sm_storage_read(), previously configured value is read from the persistent storage and the 'sm_security_mode' is populated accordingly. The 'sm_security_mode' is a SM Global variable (UCHAR), as defined in sm_extern.h.
SM Encryption Mode	This parameter configures the Encryption Mode of the Security Manager. This parameter is meaningful only if the Security Mode is other than 2. In sm_storage_write(), the current value of 'sm_encryption_mode' is written in the persistent storage. In sm_storage_read(), previously configured value is read from the persistent storage and the 'sm_encryption_mode' is populated accordingly. The 'sm_encryption_mode' is a SM Global variable (UCHAR), as defined in sm_extern.h.

SM Configuration Parameter	Porting Guide
SM Default PIN Code	This parameter configures the default PIN Code of the local Bluetooth device. In `sm_storage_write()`, the current value of '`sm_default_pin_length`' and '`sm_default_pin`' are written in the persistent storage. In `sm_storage_read()`, previously configured values are read from the persistent storage and the '`sm_default_pin_length`' and '`sm_default_pin`' are populated accordingly. The '`sm_default_pin_length`' is a SM Global variable (`UCHAR`), as defined in `sm_extern.h`. The '`sm_default_pin`' is a SM Global variable (`UCHAR` Array of size `BT_PIN_CODE_SIZE`), as defined in `sm_extern.h`. It is important to write/read the '`sm_default_pin_length`' to/from the persistent storage media before writing/reading the '`sm_default_pin`'. Bluetooth PIN Code can have a maximum of 16 octets. Since the `BT_PIN_CODE_SIZE` can be configured (in BT_limits.h) to be less than 16, it is advisable to write always 16 octets for the '`sm_default_pin`', and pad the remaining octets, if any, with 0x00. Similarly, when reading, 16 octets should be read from the persistent storage, however, only `BT_PIN_CODE_SIZE` (or, '`sm_default_pin_length`', whichever is less) number of octets should be copied to '`sm_default_pin`'.
SM Default Connection Accept	This parameter configures the default Connection Accept flag of the Security Manager. In `sm_storage_write()`, the current value of '`sm_connection_accept`' is written in the persistent storage. In `sm_storage_read()`, previously configured value is read from the persistent storage and the '`sm_connection_accept`' is populated accordingly. The '`sm_connection_accept`' is a SM Global variable (`UCHAR`), as defined in `sm_extern.h`.
SM Default Authorization	This parameter configures the default Authorization flag of the Security Manager. In `sm_storage_write()`, the current value of '`sm_authorization`' is written in the persistent storage. In `sm_storage_read()`, previously configured value is read from the persistent storage and the '`sm_authorization`' is populated accordingly. The '`sm_authorization`' is a SM Global variable (`UCHAR`), as defined in `sm_extern.h`.

SM Configuration Parameter	Porting Guide
SM 'Trusted' Device List	Devices that are marked to be 'Trusted' in the Security Manager Device Database can be stored in the persistent storage media at the time of Bluetooth OFF, and/or, shutdown of the system. The 'Trusted' Devices will be remembered and re-configured in the SM Device Database next time when Bluetooth ON is called. As of now, the Security Manager does not provide a mechanism to count number of devices that are marked to be 'trusted'. Hence, SM Device Database needs to be scanned to find this information. The Security Manager Device Database can be accessed by referencing the 'sm_devices' global variable, which is an array of SM_DEVICE_ENTITY (defined in sm_internal.h) of size SM_MAX_DEVICES (defined in BT_limits.h). The 'sm_devices' is defined in sm_extern.h. Procedure for writing/reading trusted device information is described below: (1) Write/read the number of trusted devices, from 'sm_devices' (2) For each trusted device, write/read the following information: a. Device's BD_ADDR – UCHAR array of 6 octets b. Device's Link Key – UCHAR array of 16 octets c. Device's PIN Code Length – UCHAR d. Device's PIN Code – UCHAR array of BT_PIN_CODE_SIZE octets Bluetooth PIN Code can have a maximum of 16 octets. Since the BT_PIN_CODE_SIZE can be configured (in BT_limits.h) to be less than 16, it is advisable to write always 16 octets for the device's PIN Code, and pad the remaining octets, if any, with 0x00. Similarly, when reading, 16 octets should be read from the persistent storage, however, only BT_PIN_CODE_SIZE (or, device's PIN Code length, whichever is less) number of octets should be copied to device's PIN Code.

Actual methods for opening, reading, writing and closing the persistent storage media is platform specific and out of the scope of this document. These need to be implemented for the platform, depending upon the nature of persistent storage media being used, as appropriate.

Porting EtherMind File Operations Abstraction

The EtherMind File Operations Abstraction module, consisting of BT_fops.h and BT_fops.c, are typically placed under the platforms sub-directory for a specific operating system/platform.

Porting File Abstraction Data Types

The File system specific data types are abstracted and mapped to data types used within the EtherMind Stack, and are defined in the Header file BT_fops.h. These data types may be used by the applications to make the applications platform independent. In addition, the BT_fops.h also contains function declarations of File Abstraction primitives.

Detailed description and porting guide for the data types exported by the BT_fops.h is given in the table below.

Data Type	Remarks/Porting Guide
File System Related Data Types	
BT_fops_file_handle	This data type is defined to be equivalent to the File Stream Identifier for the platform or operating system on which the File Abstraction is being ported. Example: • On Windows (User Mode) platforms, EtherMind File Abstraction Layer defines BT_fops_file_handle as FILE *, to be ANSI Conformant. It could be also defined as HANDLE and then native file operations of Windows such as CreateFile, OpenFile, ReadFile could be called from the File Abstraction Primitives described below. • On Linux (User Mode) platforms, EtherMind File Abstraction Layer defines BT_fops_file_handle as FILE *, to be ANSI Conformant. It could be also defined as int and then native file operations of Linux such as open, read, write could be called from the File Abstraction Primitives described below.

Porting File Abstraction Primitives

The file Abstraction Primitives defined below needs to be ported for the target platform to provide the desired functionality as given in the corresponding API table.

File Abstraction Primitive	Purpose/Remarks
BT_fops_get_current_directory	To get the current working directory
BT_fops_get_file_attributes	To get the file attributes of a file/directory
BT_fops_set_path_forward	To change the current working directory
BT_fops_set_path_backward	To move to the parent directory
BT_fops_create_folder	To create folder/directory
BT_fops_file_open	To open a file
BT_fops_file_write	To write to a file
BT_fops_file_read	To read from a file
BT_fops_file_close	To close a file
BT_fops_object_delete	To delete a file/directory

BT_fops_get_current_directory

Synopsis

```
API_RESULT BT_fops_get_current_directory
           (
               OUT  UCHAR    * current_directory
           );
```

IN Parameters None

OUT current_directory

Parameters This routine copies the absolute pathname of the current working directory to the array pointed by this parameter to a caller allocated "resident" memory location.

Return Value API_SUCCESS on success, and BT_FOPS_ERR_IDs defined in BT_error.h on error.

Porting Guide This primitive returns the Current Working Directory. In a non-hierarchical File System implementation it can always return "." or the absolute path of the File System.

Example:

In Linux, the getcwd() is called internally to get the Current Working Directory.

BT_fops_get_file_attributes

Synopsis

```
API_RESULT BT_fops_get_file_attributes
           (
               IN   UCHAR     * object_name,
               OUT  UINT32    * file_attribute
           );
```

IN Parameters object_name

Name of a file/folder whose attribute is requested for.

OUT Parameters file_attribute

This routine fills this 4 Octet value with the following attributes of the requested file/folder

- BT_FOPS_MASK_FOLDER (if requested object is a folder)
- BT_FOPS_MASK_FOLDER_READONLY (if requested object is a Readonly Folder)

Note: BT_FOPS_MASK_FOLDER and BT_FOPS_MASK_FOLDER_READONLY are defined in BT_fops.h

Return Value API_SUCCESS on success, and BT_FOPS_ERR_IDs defined in BT_error.h on error.

Porting Guide This primitive returns if the requested Object is a file/folder and if it is read-only.

Example:

In Linux, the stat() is called internally to get the file attributes.

BT_fops_set_path_forward

Synopsis

```
API_RESULT BT_fops_set_path_forward
           (
               IN  UCHAR    * folder_name
```

```
            );
```

IN Parameters `folder_name`

The current directory to be changed to this specified path.

OUT Parameters None

Return Value `API_SUCCESS` on success, and `BT_FOPS_ERR_ID`s defined in `BT_error.h` on error.

Porting Guide This primitive changes the current working directory to the requested path. In a non-hierarchical File System implementation it is not possible to change to a different directory and appropriate error could be returned.

Example:

In Linux, the `chdir()` is called internally to change the Current Working Directory.

BT_fops_set_path_backward

Synopsis

```
API_RESULT BT_fops_set_path_backward ( void );
```

IN Parameters None

OUT Parameters None

Return Value `API_SUCCESS` on success, and `BT_FOPS_ERR_ID`s defined in `BT_error.h` on error.

Porting Guide This primitive moves the current working directory to its parent. In a non-hierarchical File System implementation it will remain in the same directory.

Example:

In Linux, the `chdir("...")` is called internally to change the Current Working Directory.

BT_fops_create_folder

Synopsis

```
API_RESULT BT_fops_create_folder
           (
               IN  UCHAR    * folder_name
           );
```

IN Parameters `folder_name`

Name of the folder to be created.

OUT None

Parameters

Return Value API_SUCCESS on success, and BT_FOPS_ERR_IDs defined in BT_error.h on error.

Porting Guide This primitive creates a Folder of the requested name.

Example:

In Linux, the mkdir() is called internally to create the requested directory.

BT_fops_file_open

Synopsis

```
API_RESULT BT_fops_file_open
           (
               IN  UCHAR                 * file_name,
               IN  UCHAR                 * mode,
               OUT BT_fops_file_handle   * file_handle
           );
```

IN Parameters

file_name

Name of the file to be opened.

mode

This parameter points to a string beginning with one of the following sequences:

r Open file for reading. The stream is positioned at the beginning of the file.

r+ Open for reading and writing. The stream is positioned at the beginning of the file.

w Truncate file to zero length or create text file for writing. The stream is positioned at the beginning of the file.

w+ Open for reading and writing. The file is created if it does not exist. Otherwise it is truncated. The stream is positioned at the beginning of the file.

a Open for appending (writing at end of file). The file is created if it does not exist. The stream is positioned at the end of the file.

a+ Open for reading and appending (writing at end of file). The file is created if it does not exist. The stream is positioned at the end of the file.

OUT Parameters

file_handle

On success this will hold the corresponding File Stream Identifier for the platform or operating system on which the File Abstraction is being ported. Otherwise it will be set to NULL.

Return Value API_SUCCESS on success, and BT_FOPS_ERR_IDs defined in BT_error.h on error.

Porting Guide This primitive tries to open the file in the requested mode.

Example:

In Linux, the `fopen()` is called internally to open the requested file.

BT_fops_file_write

Synopsis

```
API_RESULT BT_fops_file_write
           (
               IN   UCHAR                  * buffer,
               IN   UINT16                 buf_length,
               IN   BT_fops_file_handle    file_handle,
               OUT  UINT16                 * bytes_written
           );
```

IN Parameters

`buffer`

This parameter points to the caller allocated "resident" memory location, holding the data to be written in the file.

`buf_length`

Length of the data to be written in the file. The value of this parameter could be zero, and the implementation of this API should a value of Zero as a valid case, handle appropriately, and return API_SUCCESS.

`file_handle`

This parameter is the File Stream Identifier to which the data is to be written.

OUT Parameters

`bytes_written`

Actual number of bytes written in the File (which will be less than or equal to the requested number of bytes, i.e. `buf_length`).

Return Value

`API_SUCCESS` on success, and `BT_FOPS_ERR_ID`s defined in `BT_error.h` on error.

Porting Guide

This primitive tries to write data bytes to the requested file.

Example:

In Linux, the `fwrite()` is called internally to write to the requested file.

BT_fops_file_read

Synopsis

```
API_RESULT BT_fops_file_read
           (
               IN   UCHAR                  * buffer,
               IN   UINT16                 buf_length,
               IN   BT_fops_file_handle    file_handle,
```

```
                    OUT  UINT16                 * bytes_read
                );
```

IN Parameters

`buffer`

This parameter points to the caller allocated "resident" memory location, where the read data bytes from the file is to be copied.

`buf_length`

Number of data bytes to be read from the file. The value of this parameter could be zero, and the implementation of this API should a value of Zero as a valid case, handle appropriately, and return API_SUCCESS.

`file_handle`

This parameter is the File Stream Identifier from which the data is to be read.

OUT Parameters

`bytes_read`

Actual number of bytes read from the File (which will be less than or equal to the requested number of bytes, i.e. `buf_length`).

Return Value `API_SUCCESS` on success, and `BT_FOPS_ERR_ID`s defined in `BT_error.h` on error.

Porting Guide This primitive tries to read data bytes from the requested file.

Example:

In Linux, the `fread()` is called internally to read from the requested file.

BT_fops_file_close

Synopsis

```
API_RESULT BT_fops_file_close
                (
                    IN  BT_fops_file_handle    file_handle
                );
```

IN Parameters

`file_handle`

This parameter is the Identifier of the File to be closed.

OUT Parameters None

Return Value `API_SUCCESS` on success, and `BT_FOPS_ERR_ID`s defined in `BT_error.h` on error.

Porting Guide This primitive tries to close the requested file stream.

Example:

In Linux, the `fclose()` is called internally to close the requested file.

BT_fops_object_delete

Synopsis

```
API_RESULT BT_fops_object_delete
           (
               IN  UCHAR    * object_name
           );
```

IN Parameters

`object_name`

Name of the file/folder to be deleted.

OUT Parameters

None

Return Value

`API_SUCCESS` on success, and `BT_FOPS_ERR_ID`s defined in `BT_error.h` on error.

Porting Guide

This primitive deletes a File/Folder of the requested name.

Example:

In Linux, the `remove()` is called internally to delete the requested file/directory.

Porting EtherMind Debug Library

The Debug library provides a set of APIs to facilitate the problem of logging debug messages during the development phase. The debug library can control the level of the tracing required. It also controls the output of the debug messages - console, file or serial/debug port. The debug messages are categorized as ERROR, INFORMATION and TRACE messages. Debug messages are output from the EtherMind Stack modules and these help in the diagnosis of any problems encountered during execution of the stack.

All related source code for the EtherMind Debug Library is available in the `BT_debug.c` and `BT_debug_api.h` file under `private\utils\debug\` directory.

The default implementation of EtherMind Debug Library opens a debug log file, named “debug.log”, and all debug messages are directed to this file. Depending on the platform, this implementation may require porting changes since creating “debug.log” requires a file system and sufficient storage space, which may not always be available.

The Debug Library features are routines are designed with portability in mind, and the default implementation of logging to a file can be changed to something else (such as, console/serial port) can be done easily, as discussed below.

Debug Library Primitives	Porting Guide
bt_debug_fd	This global variable, defined to be "FILE *", holds the file descriptor or handle of the opened debug log file, onto which debug messages are appended as they occur. Platforms where file system is not available, this variable can be re-defined to something else more appropriate for the platform. For example, if debug messages are required to be redirected to a serial port/device, this global variable can hold the handle or descriptor of the open serial device.
default_debug_file	This global variable, defined as CHAR array, holds the name of the log file ("debug.log") that needs to be opened for logging. Platforms where file system is not available, this variable can be re-defined to something else more appropriate for the platform. For example, if debug messages are required to be redirected to a serial port/device, this global variable can hold the name of the serial port device (such as, "COM1" on Windows, or, "/dev/ttyS0" on Linux) to be opened.
BT_Init_Debug_Library	This API is the initialization routine of the Debug Library and called from the context of BT_ethermind_init(), as the first initialization of the entire EtherMind stack. In its default implementation, this API opens, as per ANSI fopen(), the log file as described by the default_debug_file variable, and stores the handle or file descriptor of the log file in the bt_debug_fd variable. Platforms where file system is not available, this implementation can be modified to something more appropriate for the platform. For example, if debug messages are required to be redirected to a serial port/device, this routine may open the serial device, as described by the default_debug_file, and stores the device descriptor/handle in the bt_debug_fd.
BT_debug_error BT_debug_trace BT_debug_info BT_debug_dump BT_debug_direct	These debug macros are used by users of the Debug Library to output their debug messages. In its default implementation, these macros are implemented using ANSI fflush() and fprintf() calls (see BT_debug_api.h). Platforms where file system is not available, this implementation can be modified to something more appropriate for the platform. For example, if debug messages are required to be redirected to a serial port/device, these macros may write the debug messages (character strings) from stack modules onto the opened serial device.

References

Sl. No.	Reference
[1]	EtherMind Product Description Document
[2]	EtherMind Programmer's Guide
[3]	EtherMind Stack API Document, Part I & II

Contact: Bluetooth@mindtree.com
www.mindtree.com

United States

MindTree Consulting
Suite #105 #2855
Kifer Road, Santa Clara
CA 95051. USA.
Tel: +1 408 986 1000
Fax: +1 408 986 0005

Japan

Yurakucho Building 11th Floor
1-10-1, Yurakucho, Chiyoda-Ku
Tokyo, Japan 100-0006
Tel: +81 (3) 5219 2094
Fax: +81 (3) 5219 2021

United Kingdom

Regus House Windmill Hill
Business Park Whitehill Way
Swindon Wiltshire
SN5 6QR UK.
Tel: +44 (0) 1793 441418
Fax: +44 (0) 1793 441618

Singapore

Suite #12 Level 15,
Prudential Tower
30 Cecil Street
Singapore 049712.
Tel: +65 232 2751, 52, 53
Fax: +65 232 2888

India

#42 27th Cross
Banashankari II Stage
Bangalore - 560 070
Karnataka. India.
Tel: +91 80 671 1777
Fax: +91 80 671 4000