MICROTUNE™

Wireless System-On-Chip Solutions

Embedded SDK User's Guide

Company Confidential

Revision 1.2

April 23, 2002

Doc No. SPEC-00028

Abstract

The Microtune embedded Software Development Kit (SDK) allows you to modify the reference embedded applications delivered to you by Microtune and to develop your own applications based on the reference.

Approval/Date:

VP Engineering / Technology

MICROTUNE
Wireless Connectivity Division
6440 Lusk Blvd., Suite D-205
San Diego, CA 92121, USA
(Phone) 858-558-6088
(Fax) 858-558-6598
(Web) http://www.microtune.com/

Information subject to change without notice.

Legal Notices

The information in this document is believed to be accurate and reliable. Microtune assumes no responsibility for any consequences arising from the use of this information, nor from any infringement of patents or the rights of third parties which may result from its use. No license is granted by implication or otherwise under any patent or other rights of Microtune. The information in this publication replaces and supersedes all information previously supplied, and is subject to change without notice. The customer is responsible for assuring that proper design and operating safeguards are observed to minimize inherent and procedural.

Microtune assumes no responsibility for applications assistance or customer product design. The devices described in this document are not authorized for use in medical, life-support equipment, or any other application involving a potential risk of severe property or environmental damage, personal injury, or death without prior express written approval of Microtune. Any such use is understood to be entirely at the user's risk.

Microtune's products are protected by one or more of the following U.S. patents: 5,625,325; 5,648,744; 5,717,730; 5,737,035; 5,739,730; 5,805,988; 5,847,612; 6,100,761; 6,104,242; 6,144,402; 6,163,684; 6,169,569; 6,177,964; 6,218,899 and additional patents pending or filed.

Microtune, MicroTuner, Microtune logo, OneChip, Transilica and Transilica logo are trademarks of Microtune, Inc. All products and product names are the properties of Microtune, Inc. Bluetooth™ is a trademark of Bluetooth SIG, Inc. USA and licensed to Microtune, Inc. All other trademarks belong to their respective companies. Sales and corporate contact information can be found in the back of this document.

Revision History

Revision	Release Date	Author	Description
1.0	12/17/01	Bryan Batten/Brian Day/M. Hunsaker	Initial Release
1.1	01/23/02	Bryan Batten	Add development kit descriptions
1.2	04/23/02	Brian Day	Updated format and reviewed developer's additions.

Ordering Information

Model Number	Part Number	Package	Product Revision	Version

Contents

Figures

Tables

1 Introduction

The purpose of this document is to provide Microtune customers with sufficient knowledge of the program services provided to them by the Embedded Software Development Kit (SDK) to modify the included firmware applications and develop new firmware applications on the accompanying development boards.

The SDK of which this document is a part provides a development environment for a user to develop and debug custom firmware applications. It also provides sample programs illustrating the services supported by the Microtune protocol stack. Topics covered include installation, configuration, principles of operation, and integration with customer developed firmware.

1.1 System Requirements

The Microtune SDK requires the Keil 8051 Micro controller development tools Integrated Development Environment (IDE) version 2.07 or above for program compilation and linkage.

A PC development system with the following specifications is required for each module:

Table 1-1: PC Development System Requirements

Computer/Processor	Pentium® 233 MHz or higher processor
Operating System	Windows® 98/NT®/2000
Memory	16 megabytes of RAM
Hard Disk	4 megabytes of available hard-disk space
Display	VGA or better resolution monitor
Drive	CD-ROM drive
Peripherals	Microsoft® mouse or compatible pointing device
Compiler	Keil Microvision V2.20a or newer Integrated Development Environment (IDE)

1.2 SDK Contents

The Microtune embedded SDK environment consists of development boards, program development and debug facilities, a complete implementation of the Bluetooth stack, kernel services, and 8K of space available in firmware for customer applications. It contains all the necessary components to build and run complete Bluetooth firmware applications on the supplied development boards.

Three versions of the embedded SDK are available. All versions share a basic set of hardware capabilities, plus capabilities that are unique to each development kit. The development boards provided with all kits provide one RS232 serial port header connected to SIO0 of the 8051 core, one SPI port header connected to SIO1 of the 8051 core, and one JTAG header.

In addition to these basic capabilities, the MT1750-EDK provides 64KB of flash program memory with audio hardware connectivity. The MT1755-EDK provides 128KB of bank switched flash program memory with audio hardware connectivity. The MT1760-EDK provides 64KB of flash program memory with serial and USB hardware connectivity.

The following table provides a capability matrix showing the features that are unique to each kit.

Table 1-2: EDK Capability Matrix

Product Identifier	**Audio CODEC**	**USB**	**Flash Config**	**Bank Switched**	**GPIO**
MT1750-EDK	1	-	64 byte	-	28
MT1755-EDK	1	-	128 byte	Y	29
MT1760-EDK	-	Y	64 byte	-	31

Each kit contains:

- A distribution CDROM with all needed software.
- Two development boards.
- A 3.3 volt power supply with included AC power cord.
- A power connector cable.
- A USB cable.
- Two Null Modem cables.
- Two RS232 Header Adapter cables.
- A JTAG programming card.
- A printer cable to connect the PC to the JTAG programming card.
- A JTAG header adapter cable to connect the JTAG programming card to the JTAG header on the development board.
- Printed installation instructions.

The distribution CDROM contains:

- Product documentation.
- A set of Keil project (.uv2) files. These are used to build the various firmware configurations.
- A set of libraries and object files which implement the Bluetooth stack, device drivers, and kernel services.
- A set of header files defines the public interfaces of the stack and kernel software services.
- A reference implementation of the customer's firmware application in source form.
- Host PC resident JTAG based debugger back end.
- Firmware configuration tool.
- Host PC USB driver.
- Host PC virtual COM driver.

The subfolders contain the various object files and header files.

1.3 Acronyms

This section expands the various acronyms used in the document.

Table 1-3: Acronyms

Acronym	Meaning
AG	Audio Gateway
API	Application Program Interface
codec	**co**der/**dec**oder
EDK	Embedded Development Kit
GAP	Generic Access Profile
GPIO	General Purpose Input/Output
HS	Headset
IDE	Integrated Development Environment
ISO	Isochronous
ISR	Interrupt Service Routine
JTAG	Joint Test Action Group
LAP	Lower Address Part
MTU	Maximum Transmission Unit
PTE	Portable Thread Environment
SDK	Software Development Kit
SDP	Service Discovery Protocol

1.4 Organization of the Document

The rest of the document is organized as follows:

Section 2 describes how to install the product.

Section 3 provides an overview of the logical organization of the software.

Section 4 describes program operation.

Section 5 provides directions on program compilation and linkage.

Section 6 shows how to download programs to a development board and run them.

Section 7 is a service specification containing all application program interfaces (API).

Section 8 describes the files included in the distribution.

Section 9 acknowledges the contributions of "significant others".

Section 10 is a list of references.

2 Installation

This section describes how to install and configure the hardware and software components of the SDK. Topics in this section include:

- Software installation
- Configuring debug support
- Hardware configuration

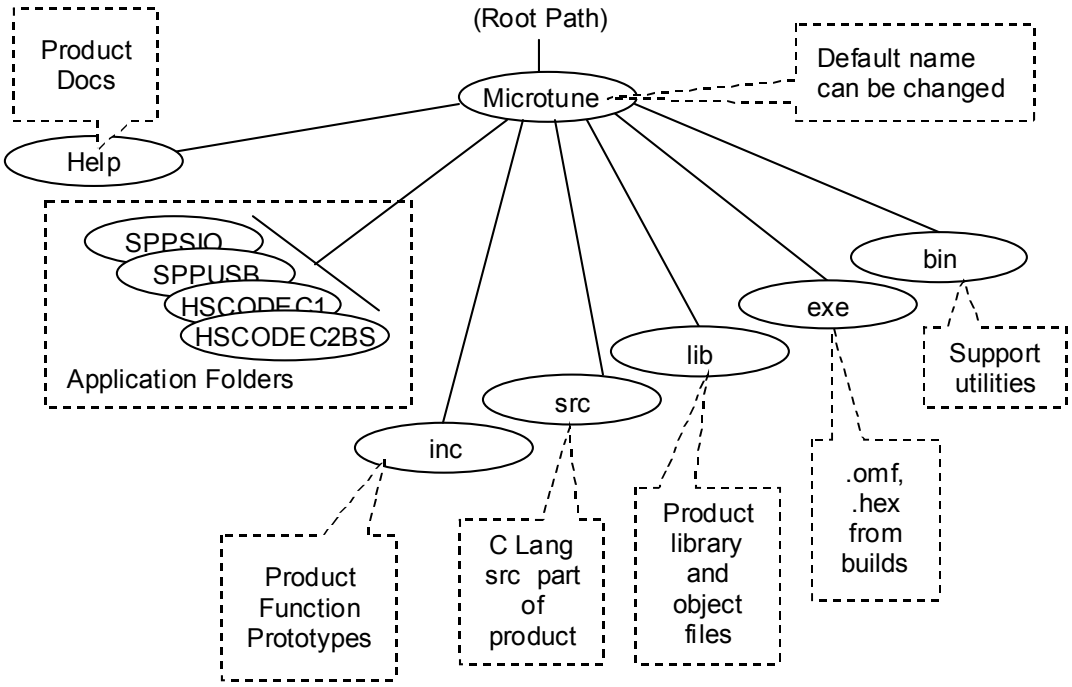
2.1 Software Installation

To install the product, insert the CDROM and wait for the installation wizard to appear. Should the installation wizard not appear, display the contents of the CDROM, then click on *setup.exe* to begin the installation process. Alternatively, you may click on the Desktop *Start* button, select *Run*, then type in the path to setup.exe on the CDROM; setup will begin when you press *Enter*.

The installation wizard begins by offering you a default folder – typically **C: \Microtune -** to install the product in. If you choose to install the product in a different folder, you will need to manually edit all the FS2 *.tcl* files that contain load directives to reflect the new path. Future versions of the kit will remove this restriction. The installation process copies all program development related files to the installation folder, then puts the components of the JTAG based debug capability into the appropriate folders of the Keil directory. Configuring Keil to use the newly installed debug components is discussed in the next section.

When the installation is complete, the installation folder contains a set of subfolders, libraries, object files, source files, and Keil project (.uv2) files. The project files contain bindings that produce software configurations to handle the various supported hardware configurations. The following figure shows the folders created by the installation process. In the figure, "Microtune" has been chosen as the name of the top-level folder in the build tree.

Figure 2-1: Installed Software Folders



A brief explanation of each folder follows:

Help: Contains all product documents in .pdf format, including this one.

Application Folders: There is one of these folders for each application. The name of each application folder reflects the nature of the specific application. In each application folder, there is a project (Keil ".uv2") file, source and header files needed for compile time binding, and all the source and header files that comprise the specific application. Clicking on the .uv2 file in one of these folders begins the compilation and link process which produces a .hex and a .omf file. By default, all listing and object files produced during compilation are placed in the application directory in which the build is done. Executables are placed in the **exe** folder (see below) by default. The following table briefly describes each application directory.

Table 2-1: Application Directories

Name	Capability
HSCODEC1	Headset Profile (HSP) using a single CODEC.
HSCODEC2BS	HSP using two CODEC's and bank switching. Uses full 128K of flash memory.
SPPSIO	Serial Port Profile (SPP) over SIO 0.
SPPUSB	SPP over USB. Requires host USB driver and virtual COM driver.

inc: Contains header files for function prototypes that define the Microtune firmware software services that are available to the application.

src: "C" language source files needed for compile time configuration.

lib: The libraries and object files that are delivered as part of the product.

exe: Provides the default location for the .omf and .hex files produced by the various application build processes.

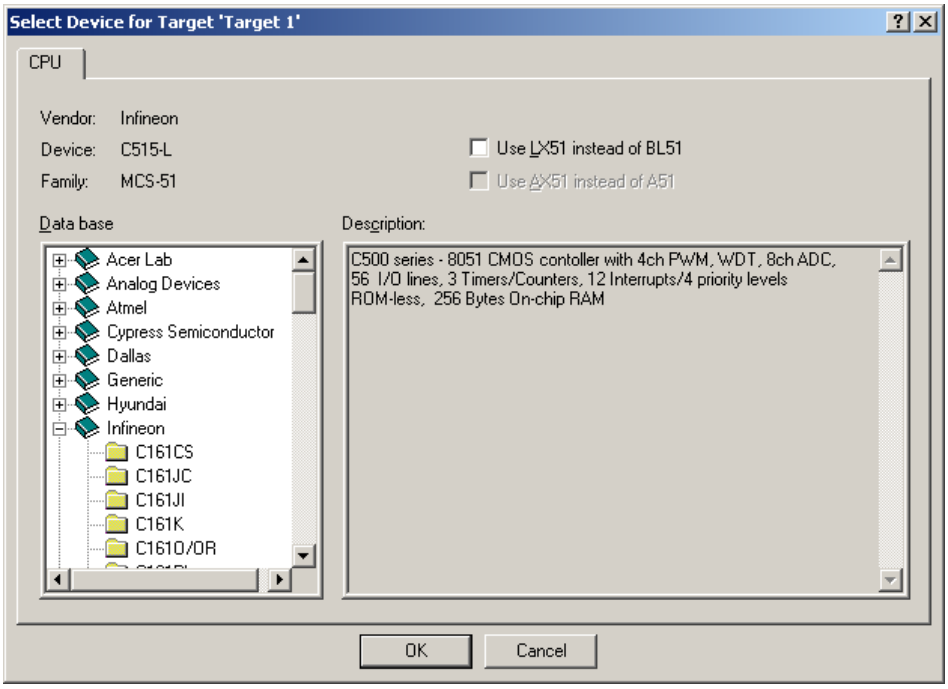
bin: Contains Microtune supplied scripts and utilities for the Windows environment.

2.2 Keil Compiler Configuration

The project files delivered as part of the SDK come with the correct 8051 device type selection. If these files are used as templates to create new applications and associated project files, no special procedures need to be taken.

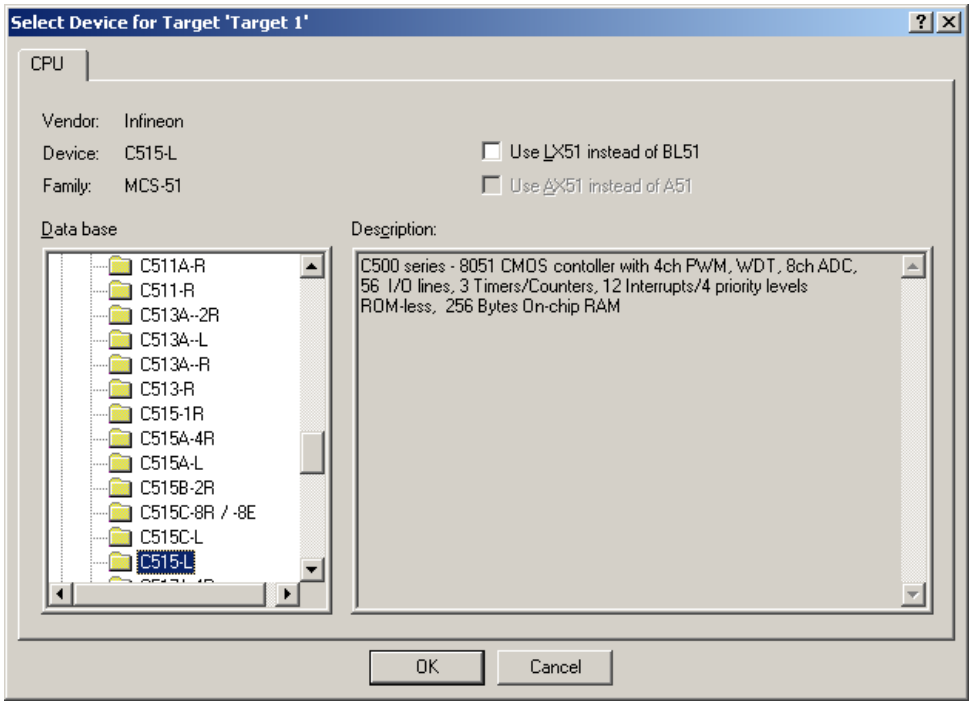
However, if you create a project file from scratch, the Keil IDE needs to be configured for the 8051 device contained in the Microtune ASIC which comes with the SDK. To do so, double click on your project file. Next, click on the "Project" menu selection. Highlight the "Options for Target `target_name'" item and click on that. The following figure shows the resulting screen dialog.

Figure 2-2: Select Device for Target Screen



Double click on the Infineon folder, then scroll down and highlight the C515-L selection as shown below:

Figure 2-3: Select Device for Target Screen (cont.)



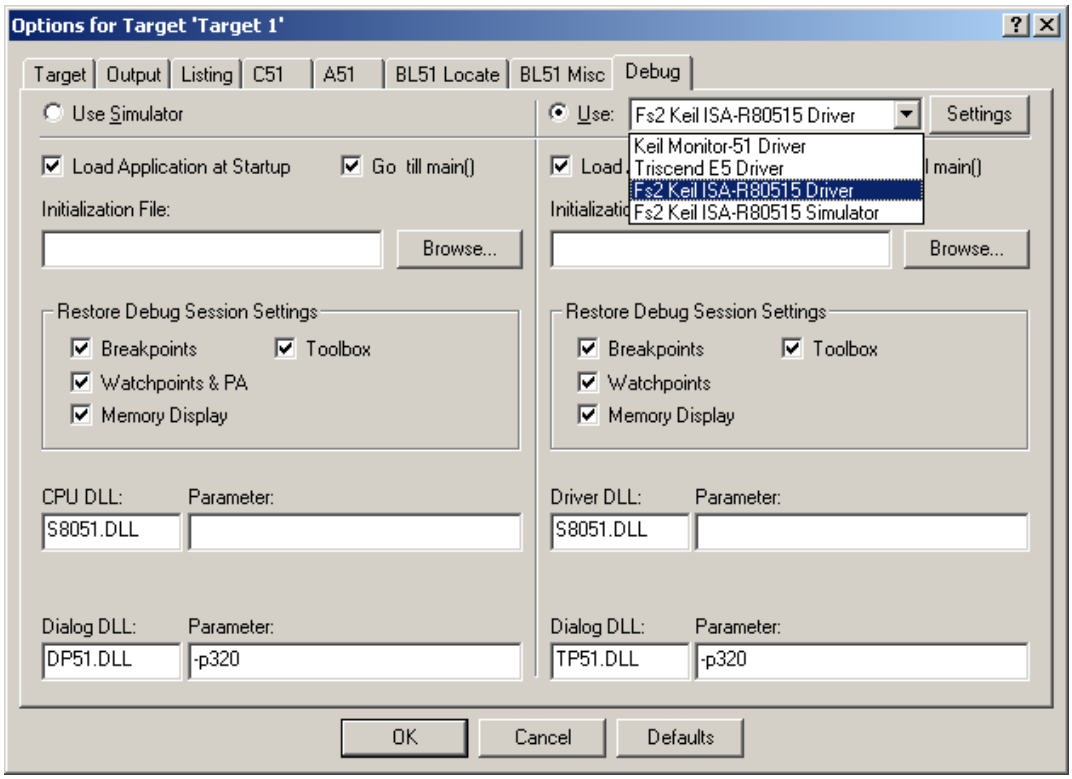
Click "OK" to complete the procedure.

2.3 Keil Debug Configuration

The project files delivered as part of the SDK come with integrated JTAG debug support. If these files are used as templates to create new applications and associated project files, no special procedures need to be taken to include integrated JTAG debug support.

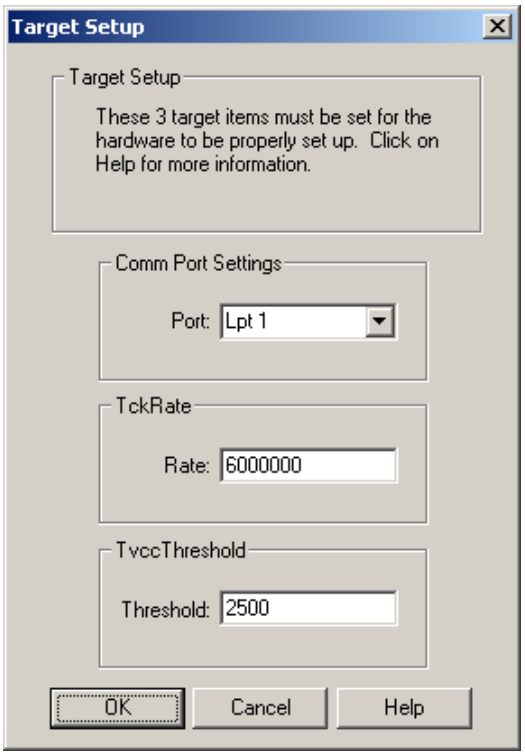
However, if you create a project file from scratch, the Keil IDE needs to be configured for the JTAG based debug support which comes with the SDK. The following figure shows the screen dialog.

Figure 2-4: Debug Option Setup Screen



Select the ISA-R80515 Driver as shown in the text box. Next, click on the "Settings" tab next to the driver menu. The following screen appears:

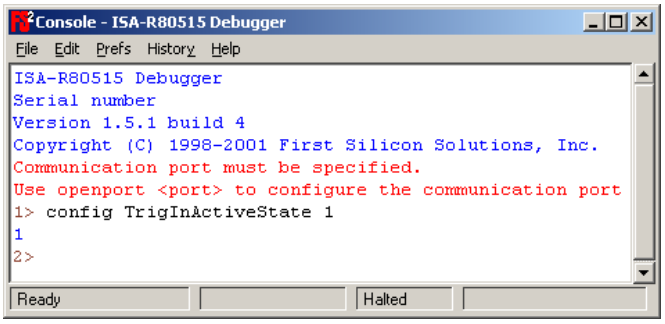
Figure 2-5: Debug Target Setup Screen



Enter the values shown for the Port, TckRate, and Threshold text boxes, then press "OK". Press "OK" again to exit the Options Window.

Next, the active logic state for the debugger's TRIG IN signal needs to be set to "1" in order to allow breakpoints to be taken. To do this, the first time you start a debugging session in Keil, click on the "Peripherals" Menu item, scroll down to the "Console" selection, and click on that. This brings up the JTAG debugger Console Window.

Figure 2-6: Setting TRIG IN Active State



Enter "config TrigInActiveState 1" to set the active logic state for the debugger's TRIG IN signal.

2.4 Hardware Configuration

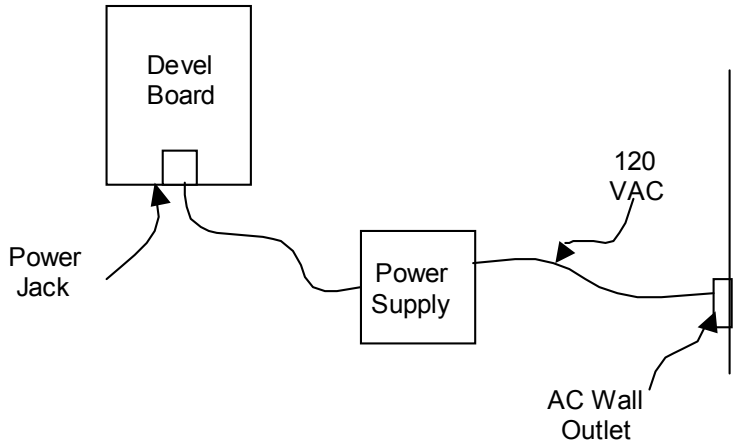
This section describes how to connect the communication and debug interfaces of a development board to the development PC.

Either the RS232 connector or the USB connector may be used as a communication interface between a development board and the development PC.

2.4.1 Power Connection

The module requires 3.0-5.5 VD. Power is supplied through a power jack. An unwired mating plug is provided for your convenience. The center contact is positive and the outer barrel ground. Connect the development board and supplied power supply as shown below.

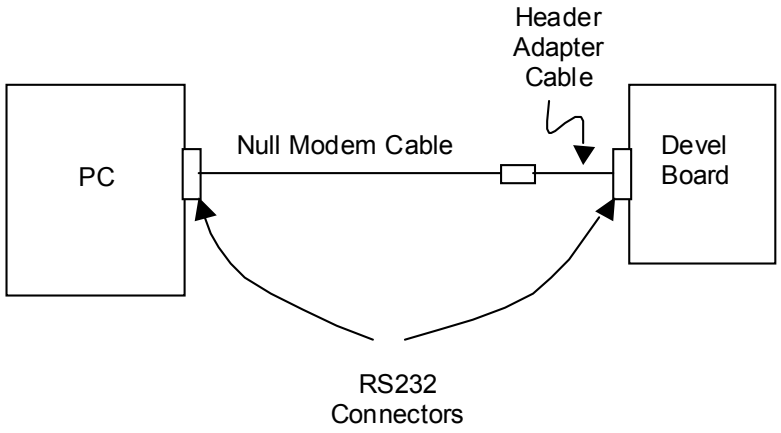
Figure 2-7: Power Connection



2.4.2 Serial PC Connection

To use the RS232 interface, connect the Microtune board to a PC through a serial port connected as illustrated below:

Figure 2-8: Serial PC Connection

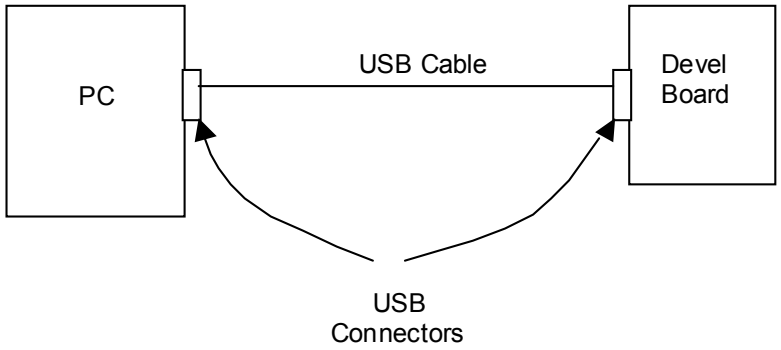


Connect the supplied RS-232 adapter cable to the module RS232 header taking care that pin #1 is properly aligned. Connect the other end of the adapter to a standard 9-pin NULL modem cable. Finally, connect the NULL modem cable to your computer.

2.4.3 USB PC Connection

To use the USB interface, connect one end of the supplied USB cable to the USB header on the development board, and the other end to a USB connector on the PC.

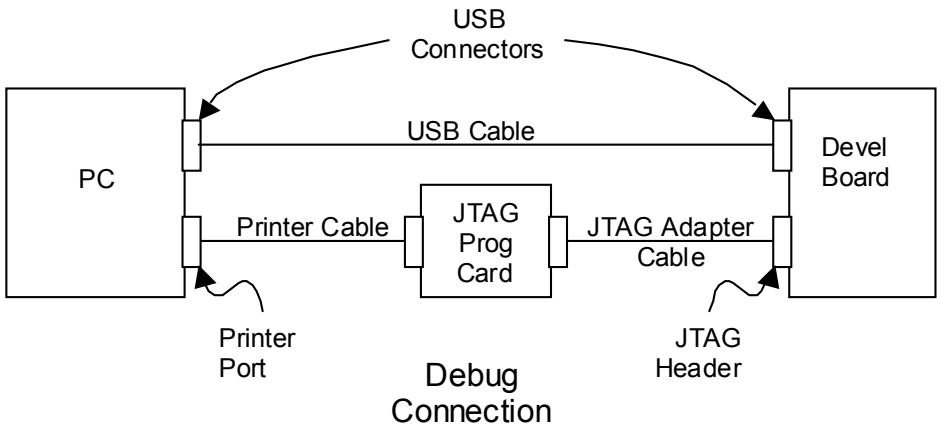
Figure 2-9: USB PC Connection



2.4.4 Debug Connection

To use the debug connection, connect the supplied JTAG adapter cable to the module JTAG header, taking care that pin #1 is properly aligned. Connect the other end of the JTAG adapter cable to the JTAG programming card. Connect the female end of the supplied printer cable to the JTAG programming card. Finally, connect the male end of the printer cable to the parallel port connector on your computer.

Figure 2-10: Debug Connection (with USB)

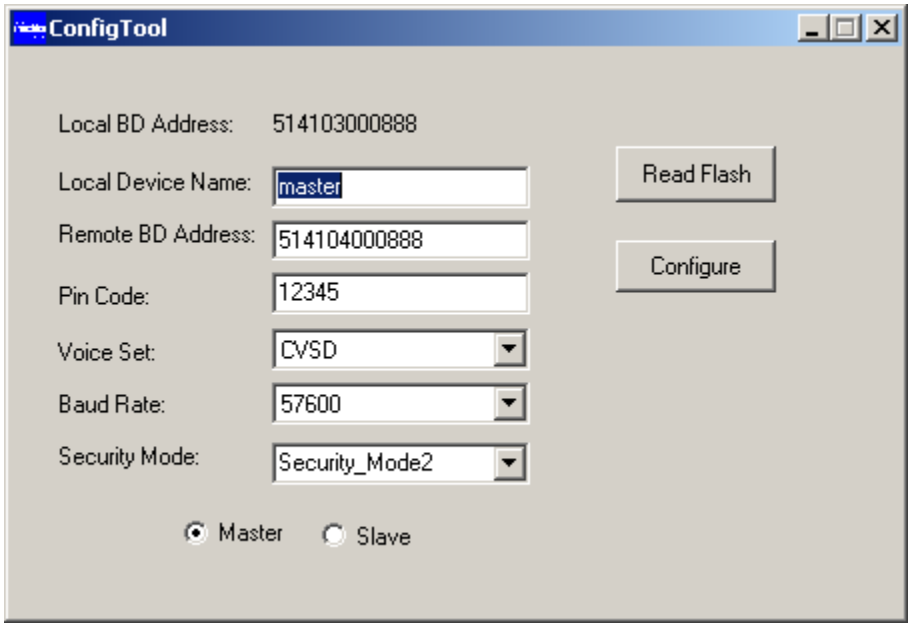


2.5 Firmware Configuration

This section deals with configuring firmware parameters. To configure firmware, a special purpose hex file must be loaded and run using **FS2**. Once this is done, the **ConfigTool** utility on the PC allows you to view and modify firmware parameters.

1. Connect COM1 on the PC to SIO port 0 on a development board using one of the supplied Null Modem cables. (See the previous section for details.)
2. Set up the debug connection as described in the previous section.
3. Run **FS2** and open the Console Window.
4. Click **File → Source**. You will see a dialog box
5. In the dialog box, navigate to <**install path**>**bin****config.tcl**, then click **Open**. This loads a special purpose hex file "**Fullstack_config.hex**".
6. Start firmware execution.
7. Run **ConfigTool**. The following window appears:

Figure 2-11: ConfigTool Window



8. Click **Read Flash** button on **ConfigTool** dialog box. The program will read and display the current settings of the module.
9. Check the Local Bluetooth address. It is used by both upper and lower stacks.
10. You may enter any optional Local Device Name as desired with up to ten alphanumeric characters.
11. You may enter any Remote board address as desired with up to 12 hex digits. It must be different from the local board address.
12. If you wish to run an application requiring a particular PIN code, you may also write in a new number. The PIN must consist of no more than 16 numeric characters.
13. The level of encryption desired selected with the drop box **Security Mode** is presently inactive and will provide selectable levels of security in the future.
14. Select either **Master** or **Slave** configuration with the selections at the bottom of the window.
15. Select which codec (**CVSD**, **A-Law**, or **U-Law**) desired for audio applications with the drop box **Voice Set**.
16. You can select the speed of the serial interface with the **Baud Rate** drop box.
17. When you are finished entering all desired changes, click on **Configure**. After the configuration is complete, the computer will inform you of success. Close the pop-up notification window.
18. Close **ConfigTool**. The module is ready for use as soon as the standard firmware is reloaded.

The module configuration may be changed as often and whenever desired.

3 Software Organization

The software supplied on the CDROM can be logically grouped into application, protocol stack, and system service components.

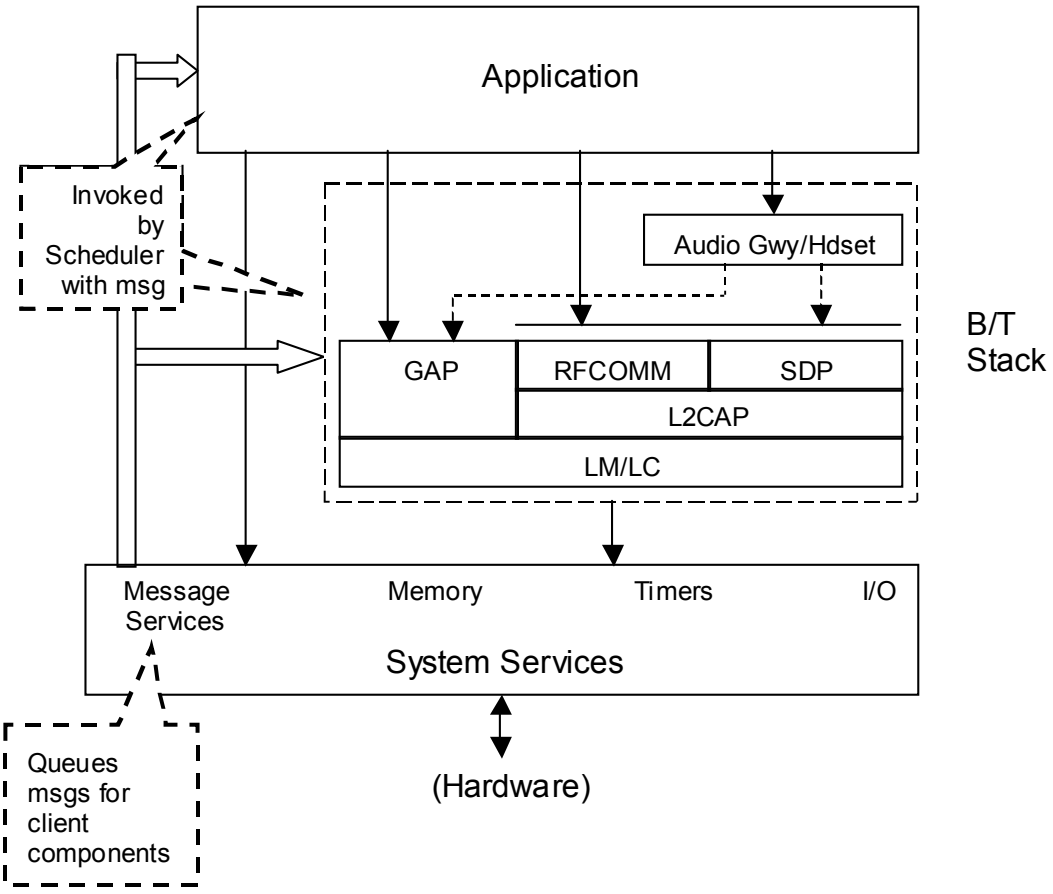
Application software determines the use to which the device is to be put; mouse, headset, etc. It uses the protocol stack and system services components to perform its function.

The Bluetooth stack components provide profiles and protocols for Audio Gateway (AG), Headset (HS), Generic Access Profile (GAP), RFCOMM, and SDP services to the application. The stack components in turn make use of the system services software component.

System services provide messaging support, I/O, memory management, and timer services that can be used by application and protocol code. An internal scheduling component determines software execution order.

The following figure shows the functional relationships among the various software components.

Figure 3-1: Software Component Relationships



These three components represent a logical grouping. The actual implementation structure, interaction, and operation are discussed in the next section.

4 Principles of Operation

All software processing depends on the services provided by Microtune's Portable Thread Environment (PTE).

4.1 Portable Thread Environment (PTE)

The PTE consists of a scheduler kernel and a set of library functions providing Thread Management, Memory Management, Timer Services and Message Services. It is an extremely small, efficient, portable, multithreading, preemptive operating system design to facilitate the deployment of complex, standards based telecommunication protocols. The PTE provides a stable and consistent shell within which its content, the communications protocol, can be ported to a wide variety of host environments – from a "bare bones" single chip standalone micro-controller host to a multi-processor host running a complex operating system.

The PTE provides:

- Small footprint – for cost sensitive applications minimum configuration requires <8K ROM, <256 RAM, 8051 core.
- Non- blocking execution model – for optimal concurrency.
- Efficient context switching – for real-time responsiveness.
- Efficient message synchronization – to reduce ISR latency.
- Cooperative multithreading – for most efficient use of CPU clock cycles.
- Preemptive multitasking – for real-time responsiveness.
- Single stack operation – for minimum RAM overhead.
- Stand-alone capability – when operating in a bare-bone environment without the aid of an external OS.
- Stable Application Framework – when plugged in as a component in a complex operating system or when running stand-alone, application software is not changed.
- Inter-task messaging – uniform inter-task messaging system makes it possible to transparently port applications between single processor and multi-processor hosts by partitioning the application at the group level.

4.2 Processing States and Priority

The PTE implements a two state machine in which all program operation occurs in either Thread State or Interrupt State. In Thread State, the basic unit of execution is the Thread. Thread State is entered when the normal processor start up sequence completes. Processing normally occurs in Thread State. In Interrupt State, the basic unit of execution is the Interrupt Service Routine (ISR). Interrupt State is entered as a result of a hardware interrupt request. Interrupt State is exited when the lowest priority ISR exits. At that time, depending on what threads have been made ready as a result of messaging activity by ISR's, execution in Thread State may or may not resume with the interrupted thread.

Within each state, multiple priorities are supported. Priorities within Thread State are determined by software. Priorities within Interrupt State are determined by hardware. The 8051 core used by Microtune provides four hardware interrupt priority levels ([4] pp. 58, 59).

Code executing in Thread State is both interruptible and preemptable. Being interruptible means that processing of the current thread is suspended whenever any hardware interrupt occurs. Preemptable means that when Thread State is resumed after the interrupt is serviced, execution may resume by starting a new thread that has been made ready as a result of a message sent by an ISR in accordance with the thread scheduling policy. When the new thread completes, the preempted thread is resumed at the point of preemption.

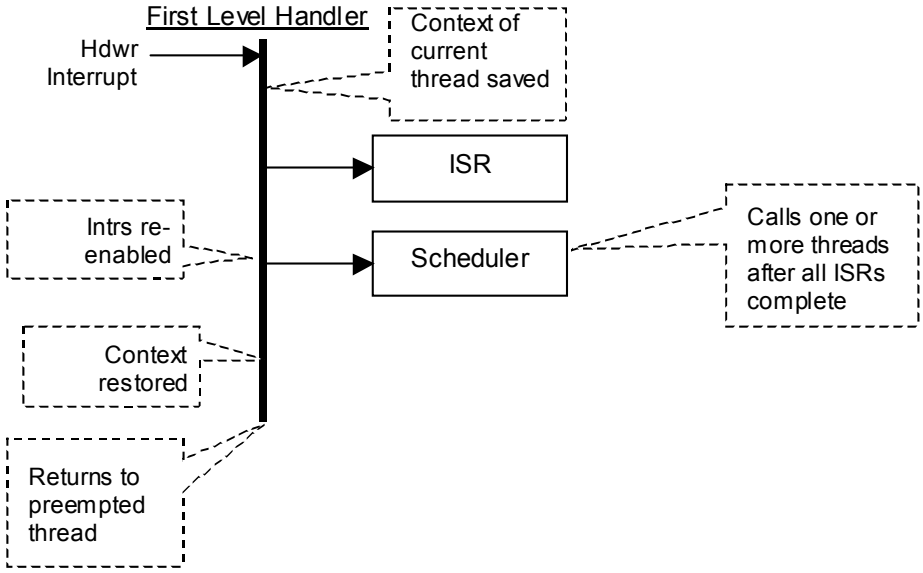
Code executing in Interrupt State is interruptible, but non- preemptable. An ISR executing at a given hardware priority can be interrupted by an ISR for a higher priority hardware interrupt. However, regardless of what threads may have been made ready as a result of processing by the higher priority ISR, execution of the interrupted ISR is guaranteed to resume at the interrupt point when the higher priority ISR exits.

ISR's are invoked by hardware events. They are responsible for handling hardware events. In order to optimize the response characteristics of the system, it is desirable to limit the time spent in Interrupt State. ISR's usually restrict themselves to capturing a snapshot of the associated hardware status, then sending a message to a thread, which can perform more extensive processing of the event without limiting the ability of the system to respond to other hardware events.

To meet the requirement for preemptive scheduling, the PTE implements a First Level Interrupt Handler. First level interrupt handling is invoked on any interrupt. All hardware interrupts are vectored to the first level handler, which saves the current register context, then calls the handler associated with the specific interrupt that has occurred. This process may nest such that the first level handler is reentered if a higher priority hardware interrupt occurs, thereby interrupting the current ISR and running the higher priority ISR. When all ISR's complete, the scheduler is called if a thread of "better" priority has been made ready as a result of an interrupt handler having sent one or more messages to threads.

When the higher priority threads (if any) complete, the scheduler returns, the register context is restored and the interrupted thread – if any – is resumed. The following figure shows the first level interrupt handling scheme.

Figure 4-1: First Level Interrupt Handling



Because the first level handler intercepts the hardware interrupts, it is important *not* to use the Keil 'interrupt' keyword if you are writing your own ISR. See the section titled "Source Level Binding" on how to write ISR's in the PTE environment.

4.3 Thread Attributes

The basic unit of execution in the PTE is the thread. Physically, a thread is a function. Threads are characterized by their PTE membership, group membership, and priority.

An application in the PTE is organized as a set of one or more cooperative groups, each group having one or more threads.

A cooperative group of threads can be used to perform the successive stages of a definable work unit, and can safely share common memory with no other synchronization mechanism regardless of their relative priorities. For example, the stages of processing needed to accept a data packet from a host PC over an RS232 connection, segment it into Bluetooth packets, then send these in succession over the air can be handled by a group of cooperating threads.

Thread message handling is serialized. Regardless of the number of messages which may be pending against a thread, and regardless of the processing state or priority of the message sender, a thread always completes handling of the current message before it is called with another message.

4.4 Message Handling

Threads are activated as a result of messages sent to them by interface functions, other threads, or by ISR's.

- Every thread in the PTE environment is uniquely identified by a *Thread ID*. Thread IDs are small numbers that are used in message routing to identify a message's destination.
- To properly route the message the message service needs to know a thread's PTE and Thread ID. These attributes are specified by the sender of a message to a thread. Selection of the destination is done by using the Thread Attribute Table (TAT). The TAT is indexed by Thread ID. It contains one entry for each thread.

Figure 4-2: Thread Attribute Table

Thread Attribute Table	
Thread[n]	*Group ID*
	Thread Priority {0-MaxPriority}
	Entry Point (Function name)

The above table diagrams one entry in the TAT.

- Group ID – Identifies group to which Thread belongs. Group may be internal (within the local PTE) or external. If internal, messages to the thread are delivered though internal message queues. If external, messages are routed to a common output queue.
- Thread ID – Identifies a specific thread within a Group.
- Entry Point – Identifies the function to be called. The function is invoked with the message number and a message buffer supplied by the sender.

Whenever a message is sent to a thread, the message information is packaged into an internal control block, which is inserted into a prioritized ready list. A thread may have any number of messages pending against it – subject to system resource limits – but in each case messages are serviced in the order they were sent. This serialization property helps threads to serve as synchronization points for the coordination of externally occurring events.

The code fragment in the following figure shows how a thread function is defined. Each time the thread responds to a message, it is called from the scheduler with the message number (`messageId`) and the address of a memory buffer (`Mbuff`) as parameters.

Figure 4-3: Sample Thread Function Definition

```
void aThread(pteMsgId_t    messageId,
             pteMBuffPtr_t pMBuff) REENTRANT
{
     switch (messageId) {
     case PTE_THREAD_INIT_MSG_ID:
          /* Do initialization here. */
          break;

     default:
          break;
     } /* End switch (messageId) */

} /* End aThread () */
```

The thread typically evaluates `messageId` to determine how to treat `pMBuff`. The `pMBuff` parameter may be any pointer sized variable. It is usually the address of the first in a list of one or more memory buffers. Each memory buffer has the following format:

Table 4-1: Memory Buffer (MBuff) Structure

Field	Size in bytes	Meaning
next	2	Address of next `Mbuff`. 0 if end of list.
length	1	Number of meaningful bytes in buff area.
offset	1	Offset to meaningful data in buff area.
buff	56	Data storage area.

The sum of the `offset` and `length` fields must be less than or equal to the size of the `buff` area.

4.5 Scheduling Policy

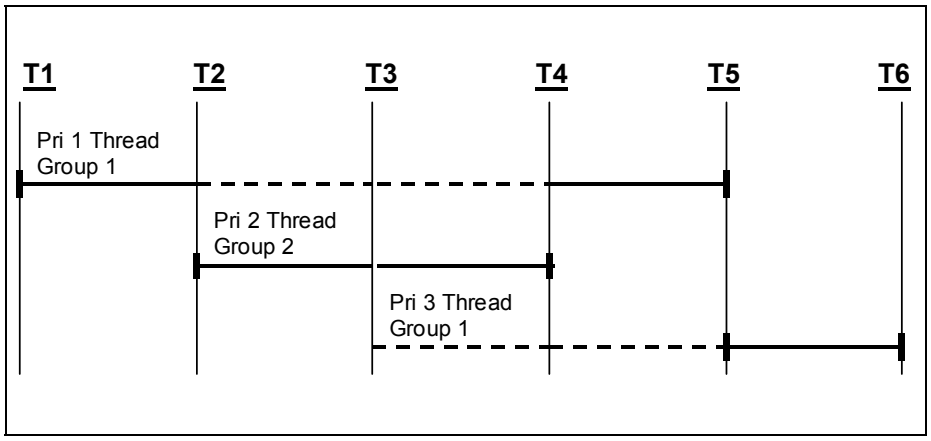
The thread is the unit of scheduling. Scheduling occurs as a side effect of sending a message to a thread. A thread is scheduled when a message is sent to it.

The PTE implements a partially preemptive scheduling policy whereby a thread of low priority is preempted when a thread of higher priority that is also a member of a different group becomes ready to run as a result of a message sent to it by an ISR. However, when a thread sends a message to another thread that is of higher priority and part of a different group, the sending thread nevertheless continues to run to completion.

Within a group, threads do not preempt each other even if they have different priorities.

The following figure illustrates the effects of the thread scheduling policy. It depicts three threads organized into two groups. Group 1 consists of two threads; one of priority 1, and the other of priority 3. Group 2 has a single priority 2 thread. In the figure, time proceeds from left to right. The start and end of a thread's execution is indicated by a short vertical bar. The time a thread spends executing is shown as a solid line segment, while the time that it is ready is shown as a dashed line segment.

Figure 4-4: Thread Scheduling Policy



At T1, a message is sent to a priority 1 thread that is part of Group 1. No other thread is ready, so thread execution begins, as indicated by the short vertical bar and the solid line.

At T2, an ISR sends a message to a priority 2 thread that is part of Group 2. When all ISR processing is done, the scheduler suspends the priority 1 thread (shown by the dashed line) that is part of Group 1, and runs the priority 2 thread of Group 2.

At T3, another ISR runs and sends a message to a priority 3 thread that is, however, part of the same group as the preempted priority 1 thread. Therefore, the priority 3 thread is simply made ready, while execution resumes with the priority 2 thread. (Had the priority 3 thread been part of a different group than the priority 1 thread, then the priority 3 thread would have run, and both the other threads would have been held in a ready state.)

At T4, the priority 2 thread completes, shown by the short vertical bar. Both threads belonging to Group 1 are now ready. However, since the priority 1 thread had already been running, it is resumed, while the priority 3 thread is held ready. Running the priority 1 thread while the priority three thread is held ready provides a mutual exclusion guarantee by ensuring that the priority 1 thread completes all modifications to data it shares with the priority 3 thread.

At T5, the priority 1 thread completes. The priority 3 thread is started.

At T6, the priority 3 thread completes. No other threads are ready to run, and the processor goes idle.

4.6 Service Model

This section describes how the basic asynchronous execution model is used to provide firmware services to client code.

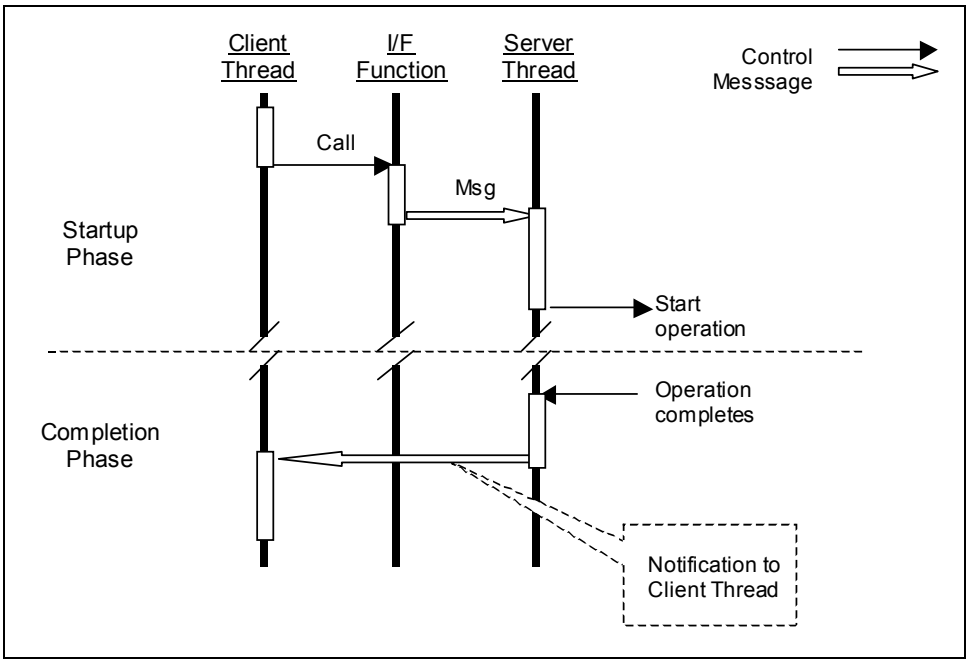
In the PTE, client program execution proceeds concurrently with the progress of a service request. Because of the non- blocking execution model, no guarantee on the state of a service request can be made when control returns to a caller – other than whether or not it was successfully begun. The actual completion of the operation occurs asynchronously. This means that on return, the actual state of the request may range from simply being queued to a request list to having actually been completed in entirety.

In this environment, the message mechanism provides the primary means of indicating the completion of a service to a client. Indications of completed services are delivered as events consisting of a message and associated memory buffer.

Given these characteristics it is useful classify service requests into a startup phase and a completion phase. The request is initiated by a call from a client thread to a service interface function. The service interface function typically packages the request into a message and sends it to a server thread, where it is processed as resources become available. The server thread then initiates the operation, such as I/O over the air link, I/O over a host link, or a timer request. Completion of the operation causes an invocation of the server thread, which in turn sends a message to the requesting client thread to indicate that the operation is complete.

The following figure shows the interactions in the PTE service model.

Figure 4-5: Service Model Interactions



The operation often involves interaction with a physical device such as a timer, R/F link, or physical link to a host PC in which some time is needed, such as the transfer of data over a link, or the expiration of a timer. In these cases, an ISR associated with the hardware sends a message to the server thread, which then performs appropriate post- processing and notifies the client thread.

The service request and completion event mechanisms may be extended to allow a client to arrange to be notified of the occurrence of a specified event or type of event which may or may not occur in the future. Examples of such arrangements are notification of a connection request from a remote node and notification of the loss of a connection.

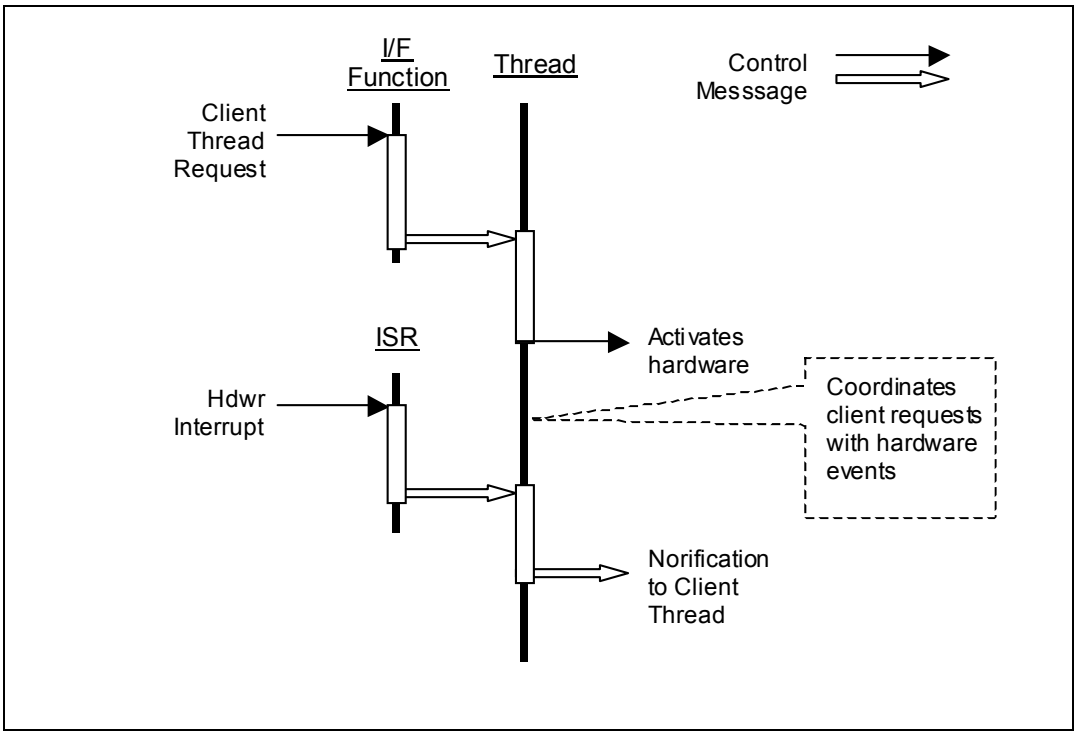
Complete descriptions of all functions and events are in the section titled "Service Specification".

4.7 Mutual Exclusion

The group concept provides a synchronization mechanism whereby threads of differing priority have exclusive access to shared variables as a result of their group membership. Because threads in the same group cannot preempt each other, regardless of their relative priorities, safe access to shared variables is provided without the overhead of conventional locks and semaphores.

Because threads process messages serially, a thread provides automatic synchronization between synchronous requests made by clients of the thread and asynchronous indications provided by hardware interrupts. The following figure shows the interaction between client requests and indications provided by hardware interrupts. The thread is responsible for managing an I/O device.

Figure 4-6: Thread/ISR Interaction



The interface function packages the client request and sends a message to the thread. If the thread is currently handling a previous message, possibly due to a previous hardware interrupt, that processing is resumed when control returns to the thread. When it does receive the message, the thread adds the request to a request list, and, if the hardware is idle, activates it. When the hardware completion event activates the ISR, it packages the device status and sends a message to the thread. The thread in turn removes the completed item from the request list, sends a notification message to the client, then begins the next operation, if any. If the thread is currently in the process of handling a client request message, the message from the ISR is deferred until the thread completes the client request. Since only the thread handles items in the request list, no critical section is needed; no locks or semaphores are used.

4.8 Protocol Services

Protocol services provide embedded applications access to full Bluetooth stack functionality. The following table enumerates the protocol and profile services that are available.

Table 4-2: Protocol Services

Component	**Description**
AGHS	Audio Gateway and Headset Profile (AGHS) services provide the protocols and procedures for connection and volume control between headset and audio gateway.
GAP	Generic Access Profile (GAP) services provide access to services provided by the lower layers of the Bluetooth stack and the security services of the upper layers.
RFCOMM	RFCOMM ("Radio Frequency COM") services provide multiple reliable channels over a single L2CAP channel and emulate multiple serial ports over a single L2CAP channel.
SDP	Service Discovery Protocol (SDP) services provide a means for applications to discover available services – such as a headset capability or dialup capability - and to determine their characteristics.
SPP	Serial Port Profile (SPP) provides serial port services to applications.

5 Creating an Application

This section describes how to compile and link your code with the Microtune embedded environment to produce firmware that may be downloaded and run on a development board. A basic knowledge of the Keil integrated development environment is assumed.

The output of the compilation and link procedures is two files, either of which may be downloaded and executed on a development board. These files usually have names of the form: App_name.*omf*, and App_Name.*hex*. The *.omf* file contains symbolic debug information used by the Keil debugger. The *.hex* file is a standard Intel hex format file ([1], [2]). By default, these files are placed in the application directory (see below).

Following subsections expand on these procedures. The section begins with a discussion of general considerations and finishes with a subsection on step-by-step procedures.

5.1 Coding Requirements

Since both threads and ISR's are invoked asynchronously, the Keil overlay mechanism cannot be relied on to properly allocate stack variables among functions. Therefore both thread functions and ISR's must be coded using the Keil `reentrant' keyword. For portability, the uppercase version `REENTRANT' may be used.

In order to provide a consistent means of saving and restoring context to meet the requirements of preemptive scheduling, the PTE uses a common first level interrupt handler. Therefore, ISR's must *not* use the Keil `interrupt' keyword. Instead, an ISR call table is provided (see **PteConfig.c**).

5.2 The Application Directory

Each application is usually assigned its own directory under the installation directory. It is customary to use a name for this directory that gives a hint of what the application itself is. The application directory is the default location for all object files and listing files produced by the compilation and link processes.

The application directory is usually set up to initially contain

- A project file
- Files needed for source level binding
- Application source files
- A **FS2** tcl script to erase flash and download a hex file.
- (Bank switched application directories only) A DOS batch file to produce a single hex file from multiple hex files.

Following subsections describe these files and how to use them.

5.3 The Project File

The project file defines what source files are to be translated into object form and combined to form a file that can be downloaded to a development board and executed. It is probably best to clone a project file that closely fits your needs from one of the reference applications, then modify the clone to suit the specific requirements of your project.

By default, project files are set up to put all object files and listing files produced by the compilation and link processes in the application directory that the project file is in. Firmware files are placed the **exe** directory by default.

5.4 Source Level Binding

To minimize code space and processing resources, threads and ISR's are bound to the embedded execution environment statically at compile time, rather than dynamically at run time. To support compile time binding, a source file – **PteConfig.c** – and a header file – **PteConfig.h** – are available in each reference application directory to use as templates. Copy the pair that most closely approximates your needs into your application directory, then modify to suit. Following paragraphs describe the changes needed to each file to perform the binding process.

The file **PteConfig.c** contains three tables; an initialization table – labeled `Pte_initTable`, an ISR table - labeled `Pte_isrJumpTable` – and a thread table – labeled `Pte_threadTable`.

The initialization table is a null terminated list of initialization functions. Functions in this table are called immediately after internal PTE initialization. Processor interrupts are enabled (EA=1), but all other bits in the IEN*x* registers are clear (disabled). All PTE services, except timer services, are available. Typically, functions in this table are responsible for initializing the various hardware components of the system. It is the responsibility of the individual hardware initialization functions in this table to set the appropriate interrupt enable bits in the IEN*x* registers. Modify the entries in this table if you are writing your own device drivers or other hardware handling initialization functions. Any messages sent to threads by initialization functions are received by the destination thread(s) only after each thread has received its thread initialization message (see below).

The ISR table entries contain references to internal and external interrupts. If you're writing your own ISRs, place their function names in the appropriate places in this table. Null entries in this table cause a software reset if the corresponding hardware interrupt occurs. As with initialization functions, timer services may not be requested by ISRs.

Threads are defined statically by entries in the thread table. Each thread you create requires an entry in the thread table. Each entry in the thread table defines the thread priority, the group to which it belongs, and the function address of the thread. The thread ID is defined by its position in the table. Since all services provided by the embedded environment are implemented as threads, care must be taken to modify only the section of the thread table marked as available for application threads. Null entries are not allowed. Each thread in the table is called with an initialization message after the system finishes calling the initialization functions. Threads receive these initialization messages before any messages sent by initialization routines.

The file **PteConfig.h** contains enumerations that define thread IDs, priorities, and groups. The values assigned to thread IDs must match the positions of the entries in the thread table. The symbolic priority and group definitions may be used for the corresponding fields of the entries in the thread table. Application thread numbers may be defined by editing the '`enum thread`' enumeration to include symbolic definitions of the application thread numbers. Since the threads that implement the services provided by the embedded environment are already defined, care must be taken to ensure that values defined for applications in the thread enumeration fall into the range reserved for application threads.

5.5 Application Source files

Application source files are the "C" language source and header files that you create to implement your application.

6 Running the Firmware

The compilation and link procedures described in the previous section produce two files, each of which may be downloaded and executed on a development board. These files usually have names of the form: App_name.*omf*, and App_Name.*hex*. The .omf file contains symbolic debug information used by the Keil debugger. The .hex file is a standard Intel hex format file ([1], [2]).

6.1 How To Build A Hex or OMF File

Here are the steps to build the hex and omf files:

1. Start Keil IDE.
2. Click **Project → Open Project**, you will see a dialog box.
3. Select <**install path**>**<app_name>.uv2** and click **Open**.
4. Click **Project → Build Target**. The hex file "app_name.hex" and omf file "app_name.omf" will be created in the **exe** directory. For a bank switched target, needed for the 128K memory environment, one hex file is produced for each specified bank. These are named app_name.h00, app_name.h01, … app_name.h*nn*.

- For a non- bank switched target, the hex or omf file may be loaded as described in the following subsection. Before the firmware for a bank switched target can be loaded using a hex file, the separate files produced for each bank must be combined into a single hex file. The following steps show how to do this.

   a. Open a Command Prompt window and navigate to the application directory.

   b. Run the batch file contained therein. This is usually named "app_name.bat". This file combines the "app_name.h*xx*" files into a single "app_name.hex" file in <**install_path**>**exe**.

6.2 How to Load A Hex or OMF File

To ensure correct operation, always make sure that the JTAG cable is connected to the module before powering up the module. When the module is connected and powered up, follow these steps to load a hex or omf file.

1. In Keil IDE, click **Start Debug**.
2. Click **Peripherals → Console Interface**.
3. When the Console Interface window opens, click **File → Source**. You will see a dialog box
4. In the dialog box, navigate to <**install path**>**bin****FlashErase.tcl**, then click **Open**. This erases the flash memory.
5. You may now close the Console Interface Window.
6. In the Keil IDE, position the mouse in the Disassembly Window and right click. You will see a menu window.
7. Select "**Load hex or object file …**". You will see a dialog box.
8. In the dialog box, go to <**install_path**>**exe****<app_name>.[hex|omf]** and click **OK**. The file will now be loaded.
9. Click **Debug → Go** to start firmware execution.

7 Service Specification

This section describes the controls and indications supplied by the Microtune embedded environment.

Controls are implemented as service requests consisting of function calls and their associated parameters.

Indications are implemented as events consisting of specific messages and the data value which accompanies each message. Indications provide information to client threads on the progress of previously initiated services and on changes in external conditions sensed by the system.

The next subsection identifies known limitations in the service. The two subsections following it describe service requests and events, respectively.

7.1 Limitations

This section identifies the known service limitations.

RFCOMM only provides five channels (not 63) per session. The Maximum Transmission Unit (MTU) is 672 bytes.

SDP only allows one session. The database is hard coded. A single request can only ask for a maximum of seven attributes at a time.

7.2 Service Requests

This section describes each function supplied by the Microtune Embedded environment in alphabetical order. Together, these functions allow you to use the Bluetooth protocol services provided by the product, perform serial I/O on the RS232 ports and USB port, request timer and memory services, and send messages between threads.

7.2.1 aghs_accept_call – Accept RING

Synopsis

```
#include "ag_hs_api.h"
void aghs_accept_call() REENTRANT;
```

Parameters

None

Description

Accept RING from audio gateway.

Return Code

None.

7.2.2 aghs_get_role – get role of aghs profile

Synopsis

```
#include "ag_hs_api.h"
void aghs_set_role(UINT8 role_setting) REENTRANT;
```

Parameters

Name	Meaning
role_setting	HEADSET or AUDIO_GATEWAY

Description

Set profile running as headset or audio gateway

Return Code

None.

7.2.3 aghs_open_audio_port – Establish SCO link

Synopsis

```
#include "ag_hs_api.h"
void aghs_open_audio_port() REENTRANT;
```

Parameters

None

Description

Establishes SCO link for voice Application should get event AGHSA_UP_SCO_CONN.

Return Code

None.

7.2.4 aghs_open_data_port – Open data channel

Synopsis

```
#include "ag_hs_api.h"
void aghs_open_data_port(UINT8 XDATA *BDAddr) REENTRANT;
```

Parameters

Name	Meaning
BDAddr	BD address

Description

Establish RFCOMM channel. Application should get event `AGHSA_UP_ACL_CONN` or `AGHSA_UP_ACL_CONN_NEG`.

Return Code

None.

7.2.5 aghs_release– release connection

Synopsis

```
#include "ag_hs_api.h"
void aghs_release() REENTRANT;
```

Parameters

None

Description

Releases SCO link and RFCOMM server channel. Application should get event `AGHSA_UP_DISC`.

Return Code

None.

7.2.6 aghs_send_ring – Send RING

Synopsis

```
#include "ag_hs_api.h"
void aghs_send_ring() REENTRANT;
```

Parameters

None

Description

Send `RING` from audio gateway.

Return Code

None.

7.2.7 aghs_set_park – enable or disable park

Synopsis

```
#include "ag_hs_api.h"
void aghs_set_park(UINT8 park_setting) REENTRANT;
```

Parameters

Name	Meaning
park_setting	0: disable park 1: enable park

Description

Enables or disables park mode

Return Code

None.

7.2.8 aghs_set_role – set role as headset or audio gateway

Synopsis

```
#include "ag_hs_api.h"
void aghs_set_role(UINT8 role_setting) REENTRANT;
```

Parameters

Name	Meaning
role_setting	HEADSET or AUDIO_GATEWAY

Description

Sets profile running as headset or audio gateway

Return Code

None.

7.2.9 aghs_transfer – transfer sound

Synopsis

```
#include "ag_hs_api.h"
void aghs_transfer() REENTRANT;
```

Parameters

None

Description

Transfers sound between audio gateway and headset

Return Code

None.

7.2.10 aghs_volume_gain – Send volume gain command

Synopsis

```
#include "ag_hs_api.h"
void aghs_volume_gain(UINT8 speakerOrMicrophone,
                      UINT8 volumeValue) REENTRANT;
```

Parameters

Name	Meaning
SpeakerOrMicrophone	For speaker or for microphone
VolumeValue	Value of volume gain

Description

Sends volume gain command. Application should get event AGHSA_UP_VGS or AGHSA_UP_VGM.

Return Code

None.

7.2.11 GAP_AddSecurityAddr – Add Security Address

Synopsis

```
#include "gap_api.h"
void GAP_AddSecurityAddr(UINT8 XDATA *BDAddr)REENTRANT;
```

Parameters

Name	Meaning
*BDAddr	XDATA pointer to the device address.

Description

This function is called to add the Bluetooth address of the remote device to the security database. This is needed for Security Mode-2.

Return Code

None

7.2.12 GAP_CancelInquiry – Cancel Inquiry

Synopsis

```
#include "gap_api.h"        /* The public interface declarations. */
void GAP_CancelInquiry()REENTRANT;
```

Parameters

None

Description

This function cancels an inquiry in progress.

Return Code

None

7.2.13 GAP_CancelPeriodicInquiry – CancelPeriodic Inquiry

Synopsis

```
#include "gap_api.h"        /* The public interface declarations. */
void GAP_CancelPeriodicInquiry()REENTRANT;
```

Parameters

None

Description

This function cancels a periodic inquiry in progress.

Return Code

None

7.2.14 GAP_DedicatedBonding – Dedicated Bonding

Synopsis

```
#include "gap_api.h"        /* The public interface declarations. */
void GAP_DedicatedBonding(UINT8 XDATA *BDAddr)REENTRANT;
```

Parameters

Name	Meaning
*BDAddr	XDATA pointer to the Device Address.

Description

This function is called to initiate the bonding procedure only to create and exchange a link key. The end of the bonding procedure is indicated by the HCIA_UP_DED_BONDING_SUCCESS or HCIA_UP_DED_BONDING_FAILURE event.

Return Code

None

7.2.15 GAP_DeviceDiscovery – Device Discovery

Synopsis

```
#include "gap_api.h"        /* The public interface declarations. */
void GAP_DeviceDiscovery()REENTRANT;
```

Parameters

None.

Description

This function is called to get the name of the first device responding to the inquiry. The name of the device is given by the `HCIA_UP_REMOTE_NAME` event to the application. This API first does inquiry and then Name Discovery of the first device.

Return Code

None

7.2.16 GAP_NameDiscovery – Name Discovery

Synopsis

```
#include "gap_api.h"        /* The public interface declarations. */
void GAP_NameDiscovery(UINT8 XDATA *BDAddr)REENTRANT;
```

Parameters

Name	Meaning
*`BDAddr`	XDATA pointer to the Device Address.

Description

This function is called to get the name of the device using the BD Address. The name of the device is given by the `HCIA_UP_REMOTE_NAME` event to the application.

Return Code

None

7.2.17 GAP_SetAuthentication – Set Authentication

Synopsis

```
#include "gap_api.h"        /* The public interface declarations. */
void GAP_SetAuthentication(BOOL mode)REENTRANT;
```

Parameters

Name	Meaning
Mode	0 for disabling the Authentication. 1 for enabling the Authentication.

Description

This function is called to enable or disable the Authentication procedure.

Return Code

None.

7.2.18 GAP_SetConnectableMode – Set Connectable Mode

Synopsis

```
#include "gap_api.h"        /* The public interface declarations. */
void GAP_SetConnectableMode(BOOL mode)REENTRANT;
```

Parameters

Name	Meaning
Mode	Value 0 - non-connectable mode and 1 - connectable mode.

Description

This API is called to set the Connectable mode.

Return Code

None

7.2.19 GAP_SetDiscoverableMode – Set Discoverable Mode

Synopsis

```
#include "gap_api.h"        /* The public interface declarations. */
void GAP_SetDiscoverableMode(BOOL mode)REENTRANT;
```

Parameters

Name	Meaning
Mode	Value 0 - non-discoverable mode and 1 - discoverable mode.

Description

This API is called to set the Discoverable mode. The Limited or General Discoverable mode depends on the Lower Address Part (LAP) value.

Return Code

None

7.2.20 GAP_SetEncryption – Set Encryption

Synopsis

```
#include "gap_api.h"        /* The public interface declarations. */
void GAP_SetEncryption(BOOL mode)REENTRANT;
```

Parameters

Name	Meaning
Mode	0 for disabling the Encryption. 1 for enabling the Encryption..

Description

This function is called to enable or disable the Encryption procedure.

Return Code

None

7.2.21 GAP_SetInquiryScanParam – Set Inquiry Scan Parameter

Synopsis

```
#include "gap_api.h"        /* The public interface declarations. */
void GAP_SetInquiryScanParam(UINT16 duration,
                UINT16 interval)REENTRANT;
```

Parameters

Name	Meaning
Duration	Amount of time for the duration of an inquiry scan. If this is 0, then default values are taken. Range: 0x0012 – 0x1000, Time = N * 0.625 msec, Range: 11.25 msec – 2560 msec the default values: N = 0x0012, Time = 11.25 msec
Interval	Amount of time between the start of 2 inquiry scan. If this is 0, then default values are taken. Range: 0x0012 – 0x1000m Time = N * 0.625 msec, Range: 11.25 – 2560 msec. The default values: N = 0x0800, Time = 1.28 Sec

Description

This function is used to set the duration of an inquiry scan and the time between start of 2 inquiry scan.

Return Code

None

7.2.22 GAP_SetLAP – Set Lower Address Part

Synopsis

```
#include "gap_api.h"        /* The public interface declarations. */
void GAP_SetLAP(UINT8 XDATA *LAP)REENTRANT;
```

Parameters

Name	Meaning
*LAP	The valid values are between 0x9E8B00– 0X9E8B3F. The value 0x9E8B33 indicates the default value (for General Discoverable Mode)

Description

This function is used to set the LAP value.

Return Code

None

7.2.23 GAP_SetPageScanParam – Set Page Scan Parameter

Synopsis

```
#include "gap_api.h"        /* The public interface declarations. */
void GAP_SetPageScanParam(UINT16 duration, UINT16 interval)REENTRANT;
```

Parameters

Name	Meaning
Duration	Amount of time for the duration of a page scan. If this is 0, then default values are taken. Range: 0x0012 – 0x1000, Time = N * 0.625 msec, Range: 11.25 msec – 2560 msec The default value: N = 0x0012 Time = 11.25 msec.
Interval	Amount of time between the start of 2 page scans. If this is 0, then default values are taken. Range: 0x0012 – 0x1000, Time = N * 0.625 msec, Range: 11.25 msec –2560 msec The default value: N = 0x0800 Time = 1.28 Sec

Description

This function is used to set the Page Scan duration and interval.

Return Code

None.

7.2.24 GAP_ SetPairableMode – Set Pairable Mode

Synopsis

```
#include "gap_api.h"        /* The public interface declarations. */
void GAP_SetPairableMode(BOOL Mode)REENTRANT;
```

Parameters

Name	Meaning
Mode	0 (default value) if pairing is not allowed and 1 if pairing is allowed.

Description

This function is used to set if pairing is allowed or not.

Return Code

None.

7.2.25 GAP_SetSecurityMode – Set Security Mode

Synopsis

```
#include "gap_api.h"        /* The public interface declarations. */
void GAP_SetSecurityMode(UINT8 Mode)REENTRANT;
```

Parameters

Name	Meaning
Mode	1 for Security Mode-1(default security mode – No Security), 2 for Security Mode-2 and 3 for Security Mode-3.

Description

This function is called to set the Security mode 1, 2 or 3. If this function is not called then, default security mode 1 is used.

Return Code

None.

7.2.26 GAP_StartInquiry – Start Inquiry

Synopsis

```
#include "gap_api.h"        /* The public interface declarations. */
void GAP_StartInquiry(UINT8 XDATA *LAP, UINT8 duration,
                UINT8 MaxResponse)REENTRANT;
```

Parameters

Name	Meaning
*LAP	Pointer to an array [3]. values between 0x9E8B00– 0X9E8B3F. This is the LAP from which the inquiry access code is derived.
Duration	Inquiry duration. Range: 0x01 – 0x30, Time = N * 1.28 sec, Range: 1.28 – 61.44 Sec
MaxResponse	Maximum number of responses to accept before exiting.

Description

This function is used to start the general/limited inquiry specifying the duration of the inquiry and the maximum number of responses. The inquiry results are given by 2 events - HCIA_UP_INQUIRY_RESULT and HCIA_UP_INQUIRY_COMPLETE to the application. For every device found, HCIA_UP_INQUIRY_RESULT event specifying the BD Address is sent to the application. When the inquiry is complete HCIA_UP_INQUIRY_COMPLETE event is sent.

Return Code

None.

7.2.27 GAP_StartPeriodicInquiry – Start Periodic Inquiry

Synopsis

```
#include "gap_api.h"        /* The public interface declarations. */
void GAP_StartPeriodicInquiry(UINT8 XDATA *LAP, UINT8 duration,
                UINT16 min_time, UINT16 max_time,
                UINT8 MaxResponse)REENTRANT;
```

Parameters

Name	Meaning
*LAP	Pointer to an array[3]. values between 0x9E8B00– 0X9E8B3F. This is the LAP from which the inquiry access code is derived.
Duration	Inquiry duration.
Min_time	Minimum amount of time between consecutive inquiries. Range:0x02–0xFFFE, Time = N * 1.28 sec, Range:2.56–83883.52 Sec
max_time	Maximum amount of time between consecutive inquiries. Range:0x03–0xFFFF, Time = N * 1.28 sec, Range:3.84–83884.8 Sec
MaxResponse	Maximum number of responses to accept before exiting.

Description

This function is used to start the general/limited Periodic inquiry. The inquiry results are given by 2 events - HCIA_UP_INQUIRY_RESULT and HCIA_UP_INQUIRY_COMPLETE to the application. For every device found, HCIA_UP_INQUIRY_RESULT event specifying the BD Address is sent to the application. When the inquiry is complete HCIA_UP_INQUIRY_COMPLETE event is sent.

Return Code

None.

7.2.28 PTE_CfgVerify – verify PTE configuration

Synopsis

```
#include "pte_core.h"       /* The public interface declarations. */
pteCfgErr_t PTE_CfgVerify(void);
```

Parameters

None.

Description

Verifies the PTE has been configured with legal values. It is a tool meant for use during development and is not needed for final production code. Must be called before passing control to PTE_Start().

Return Code

Return code is a 16-bit value with each bit signifying a different configuration error. A return value of "0" means configuration is valid. A non-zero return value indicates that at least one configuration error is present. The exact error condition(s) can be decoded using the enumeration table pteCoreConfigErrors from pte_core.h.

7.2.29 PTE_DECODE_PTE_ID – decode PTE ID from thread name

Synopsis

```
#include "pte_core.h"      /* The public interface declarations. */
#define PTE_DECODE_PTE_ID(thdName)
```

Parameters

Name	Meaning
thdName	Thread name.

Description

Macro that returns the PTE ID of a given thread name.

Return Code

The PTE ID.

7.2.30 PTE_DECODE_THREAD_ID – get thread ID from thread name

Synopsis

```
#include "pte_core.h"      /* The public interface declarations. */
#define PTE_DECODE_THREAD_ID(thdName)
```

Parameters

Name	Meaning
thdName	Thread name.

Description

Macro that returns the thread ID of a given thread name.

Return Code

The thread ID.

7.2.31 PTE_ENCODE_THREAD_NAME – builds thread name

Synopsis

```
#include "pte_core.h"      /* The public interface declarations. */
#define PTE_ENCODE_THREAD_NAME(pteId, thdId)
```

Parameters

Name	Meaning
`pteId`	The PTE ID.
`thdId`	The thread ID.

Description

Macro that returns a thread name for a give PTE ID and thread ID.

Return Code

The thread name.

7.2.32 PTE_FifoDeque – get from head of FIFO queue

Synopsis

```
#include "pte_mbuf.h"       /* The public interface declarations. */
pteMbPtr_t PTE_FifoDeque(pteFifoPtr_t pFifo) PTE_REENTRANT;
```

Parameters

Name	Meaning
`pFifo`	Pointer to FIFO queue.

Description

Remove memory buffer from head of FIFO queue. See corollary function, `PTE_FifoEnque`.

Return Code

Pointer to memory buffer removed from queue. If queue is empty, returns NULL.

7.2.33 PTE_FifoEnque – memory buffer into FIFO queue

Synopsis

```
#include "pte_mbuf.h"       /* The public interface declarations. */
void PTE_FifoEnque(pteFifoPtr_t pFifo, pteMbPtr_t pMb) PTE_REENTRANT;
```

Parameters

Name	Meaning
`pFifo`	Pointer to FIFO queue.
`pMb`	Pointer to memory buffer to place in queue.

Description

Put memory buffer into FIFO queue. It is necessary to initialize the FIFO before using it as a parameter of this function. See corollary function, `PTE_FifoDeque`.

Return Code

None.

7.2.34 PTE_FifoInit – Initialize FIFO

Synopsis

```
#include "pte_mbuf.h"       /* The public interface declarations. */
#define PTE_FifoInit(pF)
```

Parameters

Name	Meaning
`pF`	Pointer to FIFO queue.

Description

Macro that initializes a FIFO queue.

Return Code

None.

7.2.35 PTE_FifoNext – find first memory buffer in FIFO queue

Synopsis

```
#include "pte_mbuf.h"       /* The public interface declarations. */
#define PTE_FifoNext(pF)
```

Parameters

Name	Meaning
`pF`	Pointer to FIFO queue.

Description

Macro that returns pointer of first memory buffer in a given queue.

Return Code

Pointer to first memory buffer in FIFO.

7.2.36 PTE_MBuffAlloc – allocate mbuff(s)

Synopsis

```
#include "pte_mbuf.h"       /* The public interface declarations. */
pteMBuffPtr_t PTE_MBuffAlloc(pteSizeMBuff_t size) PTE_REENTRANT;
```

Parameters

Name	Meaning
Size	Size of request in bytes.

Description

Allocates an mbuff from the pool of free mbuffs. If the size requested is larger than a single mbuff, additional mbuffs are allocated and linked to accommodate the entire size. The length field of each mbuff is set to the number of allocated bytes in each mbuff.

Return Code

Pointer to allocated mbuff or mbuff chain, NULL if unable to allocate.

7.2.37 PTE_MBuffFree – free mbuff(s)

Synopsis

```
#include "pte_mbuf.h"       /* The public interface declarations. */
void PTE_MBuffFree(pteMBuffPtr_t pMBuff) PTE_REENTRANT;
```

Parameters

Name	Meaning
pMBuff	Pointer to mbuff or head mbuff chain to free

Description

Frees an mbuff or mbuff chain back to the pool of free mbuffs.

Return Code

None.

7.2.38 PTE Callbacks

The PTE provides three callbacks to handle a low memory condition, an out-of-memory condition, and the condition when the amount of available memory returns to a nominal value.

7.2.38.1 PTE_MBuffsCritical – handle critical mbuffs condition

Synopsis

```
#include "pte_mbuf.h"       /* The public interface declarations. */
void PTE_MBuffsCritical(void) PTE_REENTRANT;
```

Parameters

None.

Description

User defined function called when the number of free mbuffs falls below value set by `PTE_NUM_MBUFFS_CRITICAL` in PteConfig.h. In conjunction with related functions, it allows centralized flow control.

Return Code

None.

7.2.38.2 PTE_MbuffsExhausted – handle out of mbuffs condition

Synopsis

```
#include "pte_mbuf.h"       /* The public interface declarations. */
void PTE_MBuffsExhausted(void) PTE_REENTRANT;
```

Parameters

None.

Description

User defined function called when an mbuff is allocated but not available. In conjunction with related functions, it allows centralized flow control.

Return Code

None.

7.2.38.3 PTE_MbuffsNominal – handle return to nominal mbuffs

Synopsis

```
#include "pte_mbuf.h"       /* The public interface declarations. */
void PTE_MBuffsNominal(void) PTE_REENTRANT;
```

Parameters

None.

Description

User defined function called when the number of free mbuffs returns above value `PTE_NUM_MBUFFS_NOMINAL` after having been critical. In conjunction with related functions, it allows centralized flow control.

Return Code

None.

7.2.39 PTE_MsgRoute – routing function for external PTE's

Synopsis

```
#include "pte_core.h"       /* The public interface declarations. */
void PTE_MsgRoute(ptethdName_t thdName,
           ptemsgId_t msgId,
           pteMemPtr_t pMem) PTE_REENTRANT;
```

Parameters

Name	Meaning
`thdName`	Thread name of destination thread.
`msgId`	Message ID passed as a parameter to destination thread.
`pMem`	Two-byte pointer to XDATA memory passed as a parameter to destination thread. Alternatively, can be type cast to holding a two-byte data value. If not needed, set to NULL. *Note: in most cases, sending a memory pointer to an external PTE will not be meaningful.*

Description

This is a user-defined function for routing message to external PTE's. Any time a message is sent to an external PTE, control passes to this function. From here, the PTE ID should be decoded and the message sent on to its correct destination.

Return Code

None.

7.2.40 PTE_MsgSend – send message to a thread

Synopsis

```
#include "pte_core.h"       /* The public interface declarations. */
void PTE_MsgSend(ptethdName_t thdName,
           ptemsgId_t msgId,
           pteMemPtr_t pMem) PTE_REENTRANT;
```

Parameters

Name	Meaning
thdName	Thread name of destination thread.
msgId	Message ID passed as a parameter to destination thread.
pMem	Two-byte pointer to XDATA memory passed as a parameter to destination thread. Alternatively, can be type cast to holding a two-byte data value. If not needed, set to NULL.

Description

Places a message with the given parameters into the appropriate priority scheduling queue.

Return Code

None.

7.2.41 PTE_NumMBuffAvail – Get number of free mbuffs available

Synopsis

```
#include "pte_mbuf.h"       /* The public interface declarations. */
pteNumMBuff_t PTE_NumMBuffAvail(void) PTE_REENTRANT;
```

Parameters

None.

Description

Returns number of mbuffs available in the pool of free mbuffs.

Return Code

Number of available mbuffs.

7.2.42 PTE_Panic – Fatal error handling function

Synopsis

```
#include "pte_core.h"       /* The public interface declarations. */
void PTE_Panic(ptePanicCode_t panicCode) PTE_REENTRANT;
```

Parameters

Name	Meaning
panicCode	Predefined error code. All PTE fatal errors have a panicCode => 128.

Description

This function is a user defined fatal error handler. The PTE calls this function in case of fatal errors. It is also intended to be used by user installed code.

It is for the user to decide appropriate action for a fatal error. For typical production code, this could be a soft reset.

Return Code

None.

7.2.43 PTE_SpuriousIsr – Spurious interrupt handler

Synopsis

```
#include "pte_core.h"       /* The public interface declarations. */
void PTE_SpuriousIsr(void) PTE_REENTRANT;
```

Parameters

None.

Description

User installed function called when an undefined interrupt occurs.

Return Code

None.

7.2.44 PTE_TimerCancel – Cancel active timer

Synopsis

```
#include "pte_tmr.h"        /* The public interface declarations. */
void PTE_TimerCancel(pteTimerHandle_t handle) PTE_REENTRANT;
```

Parameters

Name	Meaning
`handle`	Handle of timer to cancel.

Description

Cancels an active timer that is no longer needed. This frees it to be reused. There is a chance timer has already expired and hence the expiration message was already sent. The user's code must account for this.

Return Code

None.

7.2.45 PTE_TimerRequest – Request timer

Synopsis

```
#include "pte_tmr.h"       /* The public interface declarations. */
pteTimerHandle_t PTE_TimerRequest(pteTimerTicks_t tickout,
                     ptemsgId_t messageId,
                     pteMemPtr_t userData) PTE_REENTRANT;
```

Parameters

Name	Meaning
`tickout`	Number of ticks until timer expires (16-bit unsigned integer).
`messageId`	Message ID to send thread once timer has expired.
`userData`	Data to be sent to thread once timer has expired. Just like the message send function, this parameter is formally a two byte pointer to XDATA memory. It can be type cast for any type of two byte data though.

Description

Requests timer that expires after given number of ticks. Upon expiration, a message is sent to the thread that requested the timer with the given message ID and two byte user data.

Timer services may not be requested by initialization functions and ISRs.

Return Code

Timer handle (must be remembered for any subsequent calls to `PTE_TimerCancel`). If no timers are available, function returns zero.

7.2.46 RFC_Close_Conn – Close RFCOMM server channel

Synopsis

```
#include "rfc_api.h"
void RFC_Close_Conn(UINT8 sessionNumber, UINT8 portNumber) REENTRANT;
```

Parameters

Name	Meaning
`sessionNumber`	RFCOMM session number
`portNumber`	RFCOMM port number of the server channel

Description

Closes a RFCOMM server channel. If the server channel is the last server channel to be closed, then RFCOMM will close the whole session automatically. The initiating side of application should expect `RFCA_UP_DISC_CNF`. The non-initiating side should expect `RFCA_UP_DISC_IND`.

Return Code

None.

7.2.47 RFC_Establish_Conn – Establish a RFCOMM server channel

Synopsis

```
#include "rfc_api.h"
void RFC_Establish_Conn(UINT8 sessionNumber,
           UINT8 serverChannelNumber, UNT8 threadId) REENTRANT;
```

Parameters

Name	Meaning
`sessionNumber`	RFCOMM session number
`serverChannelNumber`	Server channel number of RFCOMM
`threadId`	ID of thread that request the connection

Description

Open a RFCOMM server channel after RFCOMM session is opened. The initiating side of application should expect `RFCA_UP_CONN_CNF` or `RFCA_UP_CONN_CNF_NEG`. The non-initiating side should expect `RFCA_UP_CONN_IND`. The application does not to reply request of establishing server channel since RFCOMM will response automatically.

Return Code

None.

7.2.48 RFC_Open – Initiate a RFCOMM session

Synopsis

```
#include "rfc_api.h"
void RFC_Open(UINT8 XDATA *bdaddr, UINT8 threadId) REENTRANT;
```

Parameters

Name	Meaning
bdaddr	BD Address of device that will be connected to
threadId	ID of thread that request the connection

Description

Initiate a RFCOMM session connected to remove device. The initiator should expect event RFCA_UP_START_CNF or RFCA_UP_START_CNF_NEG. The non-initiator should expect RFCA_UP_START_IND. The non-initiator does not to reply the request of starting RFCOMM session since RFCOMM will response it automatically.

Return Code

None.

7.2.49 RFC_Register – Register server channel

Synopsis

```
#include "rfc_api.h"
BOOL RFC_Register(UINT8 threadId,
                  UINT8 serverChannelNumber) REENTRANT;
```

Parameters

Name	Meaning
threadId	ID of thread that request the connection
serverChannelNumber	Server channel number of RFCOMM

Description

Register a server channel in RFCOMM. Non-initiator should call this function to tell RFCOMM that the application is waiting connection request for which sever channel.

Return Code

If registration succeeded, returns true, otherwise returns false.

7.2.50 RFC_Send_Fctl – Send flow control command

Synopsis

```
#include "rfc_api.h"
void RFC_Send_Fctl(UINT8 sessionNumber, BOOL onOrOff) REENTRANT;
```

Parameters

Name	Meaning
sessionNumber	RFCOMM session number
onOrOff	0—FCON, 1—FCOFF

Description

Send flow control (FCON or FCOFF) command. The application initiating flow control command should expect RFCA_UP_FCTL_CNF. RFCOMM replies flow control command automatically

Return Code

None.

7.2.51 RFC_Send_MSC – Send MSC command

Synopsis

```
#include "rfc_api.h"
void RFC_Send_MSC(UINT8 sessionNumber, UINT8 portNumber,
                  UINT8 controlSignal, UINT8 breakSignal) REENTRANT;
```

Parameters

Name	Meaning
sessionNumber	RFCOMM session number
portNumber	RFCOMM port number
controlSignal	Control signal. If the lowest bit is 1, means no break signal
breakSignal	Break signal, not used if lowest bit of control signal is 1

Description

Send modem status command. The initiating side of application should expect RFCA_UP_MSC_CNF. The other side should expect RFCA_UP_MSC_IND. For type I device, application does not to reply MSC since RFCOMM will response automatically.

Return Code

None.

7.2.52 RFC_Send_PN – Send PN command

Synopsis

```
#include "rfc_api.h"
void RFC_Send_PN(UINT8 sessionNumber, UINT8 dlci) REENTRANT;
```

Parameters

Name	Meaning
sessionNumber	RFCOMM session number
dlci	DLCI number

Description

Send parameter negotiation command. The initiating side of application should expect RFCA_UP_PN_CNF, the other side should expect RFCA_UP_PN_IND.

Return Code

None.

7.2.53 RFC_Send_RLS – Send RLS command

Synopsis

```
#include "rfc_api.h"
void RFC_Send_RLS(UINT8 sessionNumber, UINT8 portNumber,
                  UINT8 errorCode, UINT8 errorOrStatus) REENTRANT;
```

Parameters

Name	Meaning
sessionNumber	RFCOMM session number
portNumber	RFCOMM port number
errorCode	Overrun/Parity/Framing error
errorOrStatus	0 -- no error, 1 – error

Description

Send remote line status command. The initiating side of application should expect `RFCA_UP_RLS_CNF`, the other side should expect `RFCA_UP_RLS_IND`. The application does not reply RLS since RFCOMM will respond automatically.

Return Code

None.

7.2.54 RFC_Send_RPN – Send RPN command

Synopsis

```
#include "rfc_api.h"
void RFC_Send_RPN(UINT8 sessionNumber, UINT8 dlci,
                  UINT8 RPNCmdType) REENTRANT;
```

Parameters

Name	Meaning
`sessionNumber`	RFCOMM session number
`dlci`	DLCI number
`RPNCmdType`	RPN command type: request or port value

Description

Sends remote port negotiation command to remote site. The initiating side of application should expect `RFCA_UP_RPN_CNF`, the other side should expect `RFCA_UP_RPN_IND`.

Return Code

None.

7.2.55 RFC_Send_TEST – Send TEST command

Synopsis

```
#include "rfc_api.h"
void RFC_Send_Test(UINT8 sessionNumber,UINT8 len,
                       UINT8 XDATA * testData) REENTRANT;
```

Parameters

Name	Meaning
sessionNumber	RFCOMM session number
len	Length of test data
testData	Pointer to buffer of test data

Description

Sends test command. The application initiating TEST command should expect RFCA_UP_TEST_CNF. RFCOMM will response TEST command automatically.

Return Code

None.

7.2.56 RFC_Set_LocalPortValue – Set local port value

Synopsis

```
#include "rfc_api.h"
#include "rfc_main.h"
void RFC_Set_LocalPortValue(UINT8 sessionNumber, UINT8 theDlci,
                       RPN_t XDATA* pRPNValue) REENTRANT;
```

Parameters

Name	Meaning
sessionNumber	RFCOMM session number
theDlci	DLCI number
pRPNValue	Pointer to port value

Description

Set local port values: xon character, xoff character, flow control, etc. If application doesn't call this function to set port value, RFCOMM will set it to the default value. This function can be called whenever the port setting is changed. Then the application should call RFC_Send_RPN() to inform the remote side that the port values changed.

Return Code

None.

7.2.57 RFC_Set_MTU – Set maximum frame size

Synopsis

```
#include "rfc_api.h"
void RFC_Set_MTU(UINT8 sessionNumber, UINT8 theDlci, UINT16 rfc_mtu)
                 REENTRANT;
```

Parameters

Name	Meaning
sessionNumber	RFCOMM session number
theDlci	DLCI number
rfc_mtu	Value of maximum frame size

Description

Sets maximum frame size of RFCOMM. If the application does not call this function to set maximum frame size, RFCOMM will set it to 672, the default value. This function should be called before establishing dlci. RFCOMM will ignore the changes of maximum frame size if the function is called after establishing dlci.

Return Code

None.

7.2.58 RFC_Write_Data – Send data over RFCOMM server channel

Synopsis

```
#include "rfc_api.h"
BOOL RFC_Write_Data(pteMBuffPtr_t pData,UINT8 rfcsessionNumber,
                 UINT8 rfcportNumber) REENTRANT;
```

Parameters

Name	Meaning
pData	Pointer to data
rfcsessionNumber	RFCOMM session number
rfcportNumber	RFCOMM port number

Description

Sends data over the RFCOMM server channel. If the return value is false, the application has the responsibility to free `pData`.

Return Code

1: RFCOMM places data into buffer and will send it

0: RFCOMM rejects sending data

7.2.59 SDP_CloseChannel – Close SDP Channel

Synopsis

```
#include "sdp_api.h"        /* The public interface declarations. */
bit SDP_CloseChannel()  REENTRANT;
```

Parameters

None

Description

This function is called by the application to close the SDP Channel with the remote device. After the SDP Request is completed, the application calls this function to close the SDP Channel. Otherwise the SDP Channel remains open and will not be available for other applications until closed by the SDP Server from the remote side.

The `SDPA_UP_CLOSE_SUCCESS` event is sent to the application as soon as the function is called, whether the actual SDP Channel is closed or not.

Return Code

This function return TRUE if the function succeeds and FALSE in the following cases:

a) A SDP Channel is not already open.

b) If the SDP Channel was not opened by calling the `SDP_OpenChannel()` function.

7.2.60 SDP_OpenChannel – Open SDP Channel

Synopsis

```
#include "sdp_api.h"        /* The public interface declarations. */
bit SDP_OpenChannel(ptethdName_t AppThread,
                    UINT8 XDATA *pcBTAddress)  REENTRANT;
```

Parameters

Name	Meaning
AppThread	The Application thread ID. The resulting event to the application is sent by SDP to this thread ID.
pcBTAddress	Pointer to the address of the remote device stored in XDATA. SDP opens an SDP Channel with this remote device.

Description

This function is used to open an SDP Channel. If a client wants to do an SDP Request, it should first open an SDP Channel with the remote device using this API. If a SDP Channel is established, the SDPA_UP_OPEN_SUCCESS event is sent to the application. If a SDP Channel is not established, the SDPA_UP_OPEN_FAILURE event is sent to the application.

Return Code

If a channel is already open, the function returns FALSE.

If the memory allocation for posting this request to SDP Thread fails, then this function returns FALSE.

Otherwise it returns TRUE.

7.2.61 SDP_SAReq – Attribute Request

Synopsis

```
#include "sdp_api.h"        /* The public interface declarations. */
bit SDP_SAReq(UINT32 XDATA *Handle,UINT8 AttrListLen,
           UINT8 XDATA *AttrListVal,
           SDP_ATTRIBUTE_LIST XDATA *AttrList) REENTRANT;
```

Parameters

Name	Meaning
*Handle	XDATA pointer to the Handle received from the response of ServiceSearchRequest API. This value must not be null.
AttrListLen	Number of Attributes to search for, the value should be between 1 to SDP_MAX_APP_ATTRSEQ.
*AttrListVal	Attribute list value (pointer to array of AttrListLen*2). XDATA pointer to Attribute ID's with high order first. Example: for searching 2 Attribute ID's - 0x0004 and 0x0100, it should be represented as 00 04 01 00. The Attribute ID's should be in the ascending order.
*AttrList	XDATA pointer to the structure SDP_ATTRIBUTE_LIST to store the values received from the Server. This pointer should not be NULL. The contents of this structure should be initialized to zero. The application can take the values from this structure when it receives the SDPA_UP_RESPONSE event.

The structures are defined as

```
typedef struct
{
     UINT16 AttrId;
     UINT8  AttrValLen;
     UINT8  AttrVal[SDP_MAX_APP_ATTRLEN];
} Sdp_SerAttr;

typedef struct
{
     UINT8 AttrListLen;
     Sdp_SerAttr AttrListVal[SDP_MAX_APP_ATTRSEQ];
} SDP_ATTRIBUTE_LIST;
```

Description

This API is used to make an `AttributeRequest` through an already opened SDP Channel. If any of the API's `SDP_SerChannel(), SDP_SSReq()` or `SDP_SSAReq()` was called before, then this API should be called only after the result of that API is returned to the application.

If all the returned values are received correctly `SDPA_UP_RESPONSE` is sent to the application. If the application receives `SDPA_UP_RESPONSE`, it can get the result values from the structure passed as a pointer to this API.

If the received values could not be processed correctly, `SDPA_UP_ERROR_IN_RSP` event is sent. If the `SDP_MAX_APP_ATTRLEN` value is less than any of the attribute value length in the response, then `SDPA_UP_ERROR_IN_RSP` is sent to the application. The structure can hold `SDP_MAX_APP_ATTRSEQ` number of Attributes. This is also the maximum number of Attributes that can be requested by the client. If the Attributes returned are more than this then `SDPA_UP_ERROR_IN_RSP` is sent to the application.

If the server responded with error message, `SDPA_UP_ERROR_IN_REQ` event is sent.

Return Code

This API returns TRUE if the Channel is already open. FALSE otherwise.

7.2.62 SDP_SerChannel – Get RFCOMM Server Channel

Synopsis

```
#include "sdp_api.h"        /* The public interface declarations. */
bit SDP_SerChannel(UINT16 ServiceUUID, UINT8 XDATA *Chno) REENTRANT;
```

Parameters

Name	Meaning
`ServiceUUID`	UINT16 representing the Service UUID in 16-bit type.
`Chno`	XDATA pointer to a UINT8, to store the Channel no. The application can take the channel number from this location when it receives the `SDPA_UP_RESPONSE` event.

Description

This function is used to get the Service Channel number for the ServiceUUID from the remote device. If any of the API's `SDP_SSAReq(), SDP_SSReq()` or `SDP_SAReq()` was called before, then this API should be called only after the result of that API is returned to the application. If all the returned values from the SDP Server of the remote device are received correctly by this SDP, then `SDPA_UP_RESPONSE` is sent to the application. If the received values could not be processed correctly, `SDPA_UP_ERROR_IN_RSP` event is sent. If the SDP server responded with an error message, `SDPA_UP_ERROR_IN_REQ` event is sent. If the Application receives `SDPA_UP_RESPONSE`, it can get the channel number from the location passed as the parameter `Chno`.

Return Code

TRUE if SDP Channel is already open, FALSE otherwise.

7.2.63 SDP_SSAReq – Service Search Attribute Request

Synopsis

```
#include "sdp_api.h"       /* The public interface declarations. */
bit SDP_SSAReq(UINT8 SrchPatternType,
     UINT8 SrchPatternLen,
     UINT8 XDATA *SrchPatternVal,
     UINT8 AttrListLen,
     UINT8 XDATA *AttrListVal,
     SDP_SERVICE_ATTRIBUTE_LISTS XDATA *SerAttrLists) REENTRANT;
```

Parameters

Name	Meaning
SrchPatternType	Search Pattern Type, if the UUID List (parameter *SrchPatternVal) is represented as 16bit, then this value is 2, if it is 32-bit, then this is 4 and if it is represented as 128bit, then this value is 16.
SrchPatternLen	Number of UUIDs in search pattern. This value should be > 0 and SrchPatternType*SrchPatternLen should be <= SDP_MAX_APP_SERVAL.
*SrchPatternVal	Search pattern value (pointer to array of SrchPatternType * SrchPatternLen) . XDATA pointer to UUID(s) with high order bytes first. Example: For searching for 2 UUID16 - 0x1108 and 0x1101, it should be represented as 11 08 11 01
AttrListLen	No of Attributes to search for, the value should be between 1 to SDP_MAX_APP_ATTRSEQ.
*AttrListVal	Attribute list value (pointer to array of AttrListLen*2). XDATA pointer to Attribute ID's with high order first. Example: for searching for 2 Attribute ID's - 0x0004 and 0x0100, it should be represented as 00 04 01 00. The Attribute ID's should be in the ascending order.
*SerAttrLists	XDATA pointer to structure SDP_SERVICE_ATTRIBUTE_LISTS to store the values received from the Server. This pointer should not be NULL. The contents of this structure should be initialized to zero. The application can take the values from this structure when it receives the SDPA_UP_RESPONSE event.

The structures are defined as follows:

```
typedef struct
{
      UINT16 AttrId;
      UINT8 AttrValLen;
      UINT8 AttrVal[SDP_MAX_ATTRLEN];
} Sdp_SerAttr;

typedef struct
{
      UINT8 AttrListLen;
      Sdp_SerAttr AttrListVal[SDP_MAX_ATTRSEQ];
} SDP_ATTRIBUTE_LIST;

typedef struct SERVICE_ATTR_LISTS
{
      struct SERVICE_ATTR_LISTS XDATA *next;
      SDP_ATTRIBUTE_LIST AttributeList;
} SDP_SERVICE_ATTRIBUTE_LISTS;
```

Description

This API is used to make a `ServiceSearchAttributeRequest` through an already opened SDP Channel. If any of the API's `SDP_SerChannel()`, `SDP_SSReq()` or `SDP_SAReq()` were called before, then this API should be called only after the result of that API is returned to the application.

If all the returned values are received correctly, `SDPA_UP_RESPONSE` is sent to the application. If the application receives `SDPA_UP_RESPONSE`, it can get the result values from the structure passed as a pointer to this API. If the structure can hold return values for only n records and if the return result has more than n records, then n+1th and the rest of the record values are ignored.

If the received values could not be processed correctly, `SDPA_UP_ERROR_IN_RSP` event is sent. If the `SDP_MAX_APP_ATTRLEN` value is less than any of the attribute value length in the response, then `SDPA_UP_ERROR_IN_RSP` is returned to the application. The structure can hold `SDP_MAX_APP_ATTRSEQ` number of Attributes for each record. This is also the maximum number of Attributes that can be requested by the client in `SDP_SAReq()` or `SDP_SSAReq()`. If the Attributes returned are more than this value, then an `SDPA_UP_ERROR_IN_RSP` is sent to the application.

If the server responded with error message, `SDPA_UP_ERROR_IN_REQ` event is sent.

Return Code

This API returns TRUE if the Channel is already open. FALSE otherwise.

7.2.64 SDP_SSReq - Service Search Request

Synopsis

```
#include "sdp_api.h"        /* The public interface declarations. */
bit SDP_SSReq(UINT8   SrchPatternType,
                UINT8 SrchPatternLen,
                UINT8 XDATA *SrchPatternVal,
                SDP_SERVICE_HANDLE_LIST XDATA *Handle_list)
                REENTRANT;
```

Parameters

Name	Meaning
SrchPatternType	Search Pattern Type, if the UUID List (parameter *SrchPatternVal) is represented as 16bit, then this value is 2, if it is 32bit, then this is 4 and if it is represented as 128bit, then this value is 16.
SrchPatternLen	Number of UUIDs in search pattern. This value should be > 0 and SrchPatternType*SrchPatternLen should be <= SDP_MAX_APP_SERVAL.
*SrchPatternVal	Search pattern value (pointer to array of SrchPatternType * SrchPatternLen). XDATA pointer to UUID(s) with high order bytes first. Example: For searching 2 UUID16 - 0x1108 and 0x1101, it should like 11 08 11 01
*Handle_list	XDATA pointer to structure SDP_SERVICE_HANDLE_LIST to store the values received from the Server. This pointer should not be NULL. The contents of this structure should be initialized to zero. The application can take the values from this structure when it receives the SDPA_UP_RESPONSE event.

The structure SDP_SERVICE_HANDLE_LIST is defined as

```
typedef struct
{
     UINT8 NoOfService;
     UINT32 handle[SDP_MAX_SERCLASS];
} SDP_SERVICE_HANDLE_LIST;
```

Description

This API is used to make a `ServiceSearchRequest` through an already opened SDP Channel. If any of the API's `SDP_SerChannel(), SDP_SSAReq()` or `SDP_SAReq()` were called before, then this API should be called only after the result of that API is returned to the application.

If all the returned values are received correctly, `SDPA_UP_RESPONSE` is sent to the application. If the application receives `SDPA_UP_RESPONSE`, it can get the result values from the structure passed as a pointer to this API. The number of handles that can be hold by the structure `SDP_SERVICE_HANDLE_LIST` is defined by `SDP_MAX_SERCLASS`. `SDP_MAX_SERCLASS` is also the maximum number of handles requested by the `ServiceSearchRequest` API.

If the received values could not be processed correctly, `SDPA_UP_ERROR_IN_RSP` event is sent. If the server responded with error message, the `SDPA_UP_ERROR_IN_REQ` event is sent.

Return Code

This API returns TRUE if the Channel is already open. FALSE otherwise.

7.2.65 SIO Read Callback

Synopsis

```
#include "8051sio.h"
void readcb(deviceClass_t devType, pteMBuffPtr_t pMBuff) REENTRANT;
```

Parameters

Name	Meaning
`devType`	SIO port descriptor.
`pMBuff`	Address of mbuff.

Description

The callback for SIO reads is registered with the driver by calling `SIODeviceRead()` with the callback address supplied as a parameter.

Thereafter the `Read` callback function is invoked from the serial I/O ISR whenever 20 characters are received from the host, or when any character is received in a period of 10 ms.

The mbuff parameter may be the first in a linked list of mbuffs.

The actual name of the function is defined by the user in place of the metaname given in the synopsis.

Return Code

None.

7.2.66 SIODeviceClose – SIO Close Port

Synopsis

```
#include "8051sio.h"
error_t SIODeviceClose(deviceClass_t devType) REENTRANT;
```

Parameters

Name	Meaning
devType	Device class descriptor returned by previous open request.

Description

The close function disables SIO hardware such that no further transfer of data can be done. Any read operations that are in progress are halted. All outstanding write operations are allowed to complete.

While write operations are completing, all associated read data is discarded.

Return Code

On successful completion a zero is returned. Otherwise a value of "1" is returned.

7.2.67 SIODeviceOpen – SIO Open Port

Synopsis

```
#include "8051sio.h"
error_t SIODeviceOpen(deviceClass_t devType) REENTRANT;
```

Parameters

Name	Meaning
devType	Indicates which SIO port is to be opened.

Description

The Open function prepares a SIO interface to send and receive data.

Return Code

On successful completion a zero is returned. Otherwise a value of "1" is returned.

7.2.68 SIODeviceRead – SIO Read Port

Synopsis

```
#include "8051sio.h"
error_t SIODeviceRead(deviceClass_t devType,
           void (CODE *cbfn)(pteMBuffPtr_t)) REENTRANT;
```

Parameters

Name	Meaning
devType	Specifies SIO port 0 or 1.
cbfn	Address of function to be called when SIO data arrives.

Description

The Read function accepts the address of a caller supplied callback function that is invoked when input data becomes available.

Thereafter, the callback function is invoked by the SIO read thread whenever data becomes available.

Return Code

On successful completion a zero is returned. Otherwise a value of "1" is returned.

7.2.69 SIODeviceWrite – SIO Write Port

Synopsis

```
#include "8051sio.h"
error_t SIODeviceWrite(deviceClass_t devType, pteMBuffPtr_t pMBuff)
                REENTRANT;
```

Parameters

Name	Meaning
devType	Specifies SIO port 0 or 1.
pMBuff	Address of first mbuff of data.

Description

The string of one or more mbuffs beginning with pMBuff is appended to the outgoing work list.

Return Code

If the operation is successfully initiated, a zero is returned. Otherwise a value of "1" is returned.

7.2.70 USB Callbacks

All function names given in this section are metanames. Actual names are defined by the user.

7.2.70.1 Bulk, Control, Interrupt Read Callback

Synopsis

```
#include "USBGref.h"
void readcb(epdesc_t ed, pteMBuffPtr_t pMBuff) REENTRANT;
```

Parameters

Name	Meaning
`ed`	Endpoint pair descriptor.
`pMBuff`	Address of mbuff.

Description

The callback for bulk, control, and interrupt reads is registered with the driver by calling `USBRead()` with the callback address supplied as a parameter.

Thereafter the Read callback function for a specific endpoint pair is invoked from the USB ISR whenever a data transfer from the host completes on the `OUT` member of the pair. The type of data assumed is a function of the type of endpoint on which the callback is registered.

The mbuff parameter may be the first in a linked list of mbuffs.

Return Code

None.

7.2.70.2 ISO Read Callback

Synopsis

```
#include "USBGref.h"
void isoreadcb(UINT8 XDATA *pf, UINT8 len) REENTRANT;
```

Parameters

Name	Meaning
`pf`	Address of Isochronous `OUT` data FIFO.
`len`	Unsigned length of data received.

Description

The ISO read and write callbacks are registered with the driver by calling `USBIsoRead()` with the addresses of the callbacks supplied as parameters.

The ISO read callback is called from the Isochronous `OUT` interrupt handler. The function may read up to `len` bytes of data from the FIFO address specified by `pf`.

On return from the function, the ISR flushes the FIFO, then turns it over to the USB hardware, thereby making it available for the next Isochronous OUT data transfer from the host.

Return Code

None.

7.2.70.3 ISO Write Callback

Synopsis

```
#include "USBGref.h"
void isowritecb(UINT8 XDATA *pf, UINT8 len) REENTRANT;
```

Parameters

Name	Meaning
`pf`	Address of Isochronous `IN` data FIFO.
`len`	Unsigned size of `IN` FIFO.

Description

The ISO read and write callbacks are registered with the driver by calling `USBIsoRead()` with the addresses of the callbacks supplied as parameters.

The ISO write callback is called from the Isochronous IN interrupt handler. The function may write up to `len` bytes of data to the FIFO address specified by `pf`.

On return from the function, the ISR turns the FIFO over to the USB hardware only if any data has been written to it, thereby initiating the next Isochronous IN data transfer.

The available FIFO size is determined by the current alternate setting for the second interface.

The ISO `IN` data stream must be initiated by a call to `USBIsoWrite()`.

Return Code

None.

7.2.71 USBClose – USB Close Endpoint

Synopsis

```
#include "USBGref.h"
error_t USBClose(epdesc_t ed) REENTRANT;
```

Parameters

Name	Meaning
ed	Endpoint descriptor in range 0 – 3 returned by previous open request.

Description

The close function disables USB hardware such that no further transfer of data over the bus can be done on the specified endpoint pair. Any read operations that are in progress are halted. All outstanding write operations are allowed to complete.

For the control endpoint, only class type requests are affected. Standard type requests continue to be processed by the driver.

Return Code

On successful completion a zero is returned. Otherwise a value of "1" is returned.

7.2.72 USBIoctl – USB I/O Control

Synopsis

```
#include "USBGref.h"
INT8 USBIoctl(UINT8 IOCtrlCode, UINT8 iParm) REENTRANT;
```

Parameters

Name	Meaning
`IOCtrlCode`	Specifies the control function to be performed.
`iParm`	Additional parameter, depending on the specific control function.

The following I/O control code values are defined for the `ioControl` parameter in **USBGref.h**:

I/O Control Code	Meaning
USB_IOCReqStatusChange	Request Status Change Message. The following parameter is the message to be sent to the current thread on USB status change.
USB_IOCResume	Resume. Directs the USB driver to begin remote wakeup (i.e. locally initiated resume) signaling. Second parameter is null.
USB_IOCSetStall	Set Stall condition. Endpoint specified by second parameter will respond to IN tokens with STALL handshake.
USB_IOCClrStall	Clear Stall condition. Endpoint specified by second parameter will respond to IN tokens with NAK or data.
USB_IOCGetPktSize	Get packet size. Returns packet size for endpoint specified by second parameter.
USB_IOCSendNullPkt	Send Null packet. Sends zero length packet on endpoint specified by second parameter in response to the next IN Token.
USB_IOCFlushIn	Flush IN. Empties IN FIFO, empties transmit queue on endpoint specified by second parameter.
USB_IOCFlushOut	Flush OUT. Empties OUT FIFO, releases receive queue on endpoint specified by second parameter.
USB_IOCSetISOBlkFactor	Set ISO Blocking Factor. Second parameter specifies ISO transfer size in multiples of the current ISO packet size. Default is 3.
USB_IOCFlowOn	Flow On. NAK OUT transfers on endpoint specified by second parameter.
USB_IOCFlowOff	Flow Off. ACK OUT transfers on endpoint specified by second parameter.

Description

The I/O control function allows the caller to specify USB parameters such as endpoint functionality and quality of service in a manner that is independent of the specific controller hardware in use. The advantage of this approach is that hardware dependencies are well localized so that different hardware can be used as technology improves with a minimal impact on software.

Return Code

On successful completion of all requests except USB_IOCGetPktSize, a zero is returned.

When the I/O Control code is USB_IOCGetPktSize, the return value is the packet size in bytes.

On error, a –1 is returned.

7.2.73 USBIsoRead – USB Isochronous Read Endpoint

Synopsis

```
#include "USBGref.h"
error_t USBIsoRead(epdesc_t ed, void
           (CODE *cbfn)(epdesc_t, pteMBuffPtr_t),
           void (CODE *wcbfn)(epdesc_t, pteMBuffPtr_t)) REENTRANT;
```

Parameters

Name	Meaning
`ed`	Endpoint descriptor in range 0 – 3 returned by previous open request.
`cbfn`	Read callback to be called when USB `OUT` data arrives.
`wcbfn`	Write callback to be called when USB `IN` data completes.

Description

The ISO Read function accepts the addresses of caller supplied read and write callback functions that are invoked when input data becomes available on the OUT member of the specified endpoint pair and when data transfer completes on the `IN` member of the specified endpoint pair.

Thereafter, the read callback is called from the USB ISR each time an ISO `OUT` data arrives, and the write callback is called from the USB ISR each time an ISO `IN` data transaction completes.

Return Code

On successful completion a zero is returned. Otherwise a value of "1" is returned.

7.2.74 USBIsoWrite – USB Isochronous Write Endpoint

Synopsis

```
#include "USBGref.h"
error_t USBIsoWrite(UINT8 XDATA *pb, UINT8 len) REENTRANT;
```

Parameters

Name	Meaning
pb	Pointer to buffer in external data memory.
len	Unsigned length.

Description

This function is used to initiate Isochronous IN data transfers.

If the USB device has been opened, and is not active, up to len bytes of external data beginning at pb are written to the Isochronous IN endpoint FIFO. The driver then initiates actual transfer of the data over the USB. If the size of the request exceeds the size of the FIFO, then only that many bytes are transferred.

Once the first ISO packet is started, subsequent packets can be written using the ISO write callback function.

The available FIFO size is determined by the current alternate setting for the second interface.

Return Code

If the operation is successfully initiated, the number of bytes actually written is returned. This is the lesser of the amount actually requested and the FIFO size.

If the USB device is not open, or if it is currently transferring Isochronous IN data, a value of "1" is returned.

7.2.75 USBOpen – USB Open Endpoint Type

Synopsis

```
#include "USBGref.h"
epdesc_t USBOpen(USB_EPType ep) REENTRANT;
```

Parameters

Name	Meaning
ep	Endpoint type indicator in range 0 – 3. Indicates one of Control, Interrupt, Bulk, or Isochronous.

Description

The `Open` function prepares a USB interface endpoint pair of the type defined by the `ep` parameter to accept data from the bus on the `OUT` member of the pair, and send data over the bus on the `IN` member of the pair. Where the endpoint type specifies the control endpoint, only the single bidirectional control endpoint is activated.

If the request is successful, an endpoint descriptor is returned to the caller to be used in subsequent write, read, I/O control, and close requests.

Return Code

On successful completion an endpoint descriptor in the range 0 – 3 is returned. Otherwise a value of "1" is returned.

7.2.76 USBRead – USB Control, Bulk, Interrupt Read Endpoint

Synopsis

```
#include "USBGref.h"
error_t USBRead(epdesc_t ed,
           void (CODE *cbfn)(epdesc_t, pteMBuffPtr_t)) REENTRANT;
```

Parameters

Name	Meaning
`ed`	Endpoint descriptor in range 0 – 3 returned by previous open request.
cbfn	Address of function to be called when USB data arrives.

Description

The Read function accepts the address of a caller supplied callback function that is invoked when input data becomes available on the OUT member of the specified endpoint.

Thereafter, the callback function is invoked whenever data becomes available on the associated endpoint.

Return Code

On successful completion a zero is returned. Otherwise a value of "1" is returned.

7.2.77 USBWrite – USB Control, Bulk, Interrupt Write Endpoint

Synopsis

```
#include "USBGref.h"
error_t USBWrite(epdesc_t ed, pteMBuffPtr_t mp) REENTRANT;
```

Parameters

Name	Meaning
ed	Endpoint descriptor in range 0 – 3 returned by previous open request.
mp	Address of first mbuff of data.

Description

The string of one or more mbuffs beginning with mp is appended to the outgoing transmit queue.

Return Code

If the operation is successfully initiated, a zero is returned. Otherwise a value of "1" is returned.

7.3 Events

This section describes each event generated by the Microtune Embedded environment in alphabetical order. Events are used to provide indications on the state of requested services and on changes in external conditions sensed by the system.

The header file **upperlayer_events.h** has all AG, HS, RFCOMM, GAP, and SDP event message definitions. The header file **pte_core.h** defines the PTE event message.

7.3.1 AGHSA_UP_ACL_CONN – ACL Connection Is Up

Event Data

None.

Description

Indication of ACL connection is up.

7.3.2 AGHSA_UP_ACL_CONN_NEG – ACL Connection Is Failed

Event Data

None.

Description

Indication of ACL connection establishment is failed.

7.3.3 AGHSA_UP_ACCEPT_RING – Headset Accepts Ring

Event Data

None.

Description

Indication of Headset accepting `RING`. Only `AudioGateway` has this event.

7.3.4 AGHSA_UP_DISC – Indication Of Connection Release

Event Data

None.

Description

Indication of connection release.

7.3.5 AGHSA_UP_RING – Indication Of RING Command

Event Data

None.

Description

Indication of `RING` command from `AudioGateway`. Only Headset has this event.

7.3.6 AGHSA_UP_SCO_CONN – SCO Connection Is Up

Event Data

None.

Description

Indication that the SCO connection is up.

7.3.7 AGHSA_UP_VGS – Volume Gain For Speaker

Event Data

UINT8

Value of volume gain.

Description

Indication of volume gain for speaker.

7.3.8 AGHSA_UP_VGM – Volume Gain For Microphone

Event Data

UINT8

Value of volume gain.

Description

Indication of volume gain for Microphone.

7.3.9 HCIA_UP_DED_BONDING_FAILURE – Bonding Failed

Event Data

None.

Description

This event is sent when the API `GAP_DedicatedBonding()` is called and the bonding procedure fails.

7.3.10 HCIA_UP_DED_BONDING_ SUCCESS – Bonding Success

Event Data

None.

Description

This event is sent when the API `GAP_DedicatedBonding()` is called and the bonding procedure succeeds.

7.3.11 HCIA_UP_INQUIRY_COMPLETE – Inquiry is completed

Event Data

None.

Description

This event is sent as a result of calling the API `GAP_StartInquiry()` or the API `GAP_StartPeriodicInquiry()`. If an inquiry is initiated and as and when each device is discovered the event `HCIA_UP_INQUIRY_RESULT` is sent. If the specified number of device is found or if the time elapses, then the completion of the inquiry is notified by `HCIA_UP_INQUIRY_COMPLETE` event.

7.3.12 HCIA_UP_INQUIRY_RESULT – Result of an Inquiry

Event Data

Pointer to Mbuff.

Mbuff->buff[*n*]	Description
0	Event Code – Value is always 0x02.
1	Parameter length – Depends on the number of Inquiry responses. Since the Inquiry response value is always 1, this value is always 15.
2	Number of Inquiry responses – always 1.
3-8	Bluetooth device address of the discovered device.
9	`Page_Scan_Repetition_Mode` – Value 0x00 for R0, 0x01 for R1, 0x02 for R2.
10	`Page_Scan_Period_Mode` – Value 0x00 for P0, 0x01 for P1, 0x02 for P2.
11	`Page_Scan_Mode` – Value 0x00 for Mandatory Page Scan Mode, 0x01 for Optional Page Scan Mode 1, 0x02 for Optional Page Scan Mode 2, 0x03 for Optional Page Scan Mode 3.
12,13,14	Class of Device
15, 16	Clock Offset

Description

This event may be sent as a result of calling the API `GAP_StartInquiry()` or the API `GAP_StartPeriodicInquiry()`. If an inquiry is initiated as and when each device is discovered the event `HCIA_UP_INQUIRY_RESULT` is sent. If no device is discovered, then this event is not sent at all.

7.3.13 HCIA_UP_REMOTE_NAME – Name of the Remote Device

Event Data

Pointer to Mbuff.

Mbuff->buff[*n*]	Description
0	Length of the Name
1-n	Remote Name

Description

This event is sent as a result of calling the API `GAP_NameDiscovery()` or the API `GAP_DeviceDiscovery()`.

7.3.14 RFCA_TRS_SESSION_DOWN – session down

Event Data

UINT16

RFCOMM session number. The session number can range in value from 0 to 1.

Description

The indicated session has gone down by itself

7.3.15 PTE_THREAD_INIT_MSG_ID – PTE thread initialize event

Event Data

None.

Description

Each thread receives this event when PTE is first started.

7.3.16 PTE Timer Event – PTE Timer Expired

Event Data

UINT16

The actual value is specified by the caller in the `PTE_TimerRequest()` call.

Description

This event is sent to the requesting thread when a PTE timer expires. The actual value of the PTE timer event message is specified by the caller in the `PTE_TimerRequest()` call.

`PTE_TimerCancel()` must be called to free the timer handle.

7.3.17 RFCA_UP_CONN_IND – Connection Indication

Event Data

Pointer to Mbuff.

Mbuff->buff[*n*]	Description
0	RFCOMM session number.
1	RFCOMM port number.
2,3	The least 12 significant bits are ACL connection ID.

Description

Indication of connection establishment request for server channel. RFCOMM will reply to the request by itself.

7.3.18 RFCA_UP_CONN_CNF – Connection Confirmation

Event Data

Pointer to Mbuff.

Mbuff->buff[*n*]	Description
0	RFCOMM session number.
1	RFCOMM port number.
2,3	The least 12 significant bits are ACL connection ID.

Description

Confirmation of successful connection establishment for server channel.

7.3.19 RFCA_UP_CONN_CNF_NEG – Connection Negative Confirm

Event Data

Pointer to Mbuff.

Mbuff->buff[*n*]	Description
0	RFCOMM session number.
1	RFCOMM port number.

Description

Confirmation of failed connection establishment for sever channel.

7.3.20 RFCA_UP_DATA_IND – Data Indication

Event Data

Pointer to list of one or more Mbuffs. The first mbuff has the information shown in the following table:

Mbuff->buff[*n*]	Description
0	RFCOMM session number.
1	RFCOMM port number.
Offset thru offset+length	Data starts at position defined by `mbuff->offset` field, and continues for `mbuff->length` bytes.

Following mbuffs in the list (if any) consist entirely of data. The start of the data in each following mbuff is defined by the offset field (always 0), and the amount of data is defined by the length field.

Description

Indication of data received by RFCOMM.

7.3.21 RFCA_UP_DISC_IND – Disconnect Indication

Event Data

Pointer to Mbuff.

Mbuff->buff[*n*]	Description
0	RFCOMM session number.
1	RFCOMM port number.

Description

Indication of RFCOMM server channel disconnect request. RFCOMM will reply automatically.

7.3.22 RFCA_UP_DISC_CNF – Disconnect Confirmation

Event Data

Pointer to Mbuff.

Mbuff->buff[*n*]	Description
0	RFCOMM session number.
1	RFCOMM port number.

Description

Confirmation of RFCOMM server channel disconnect request.

7.3.23 RFCA_UP_FCTL_CNF – Flow Control Command Confirmation

Event Data

Pointer to Mbuff.

Mbuff->buff[*n*]	Description
0	RFCOMM session number.

Description

Confirmation of RFCOMM flow control command.

7.3.24 RFCA_UP_MSC_IND – MSC Command Indication

Event Data

Pointer to Mbuff.

Mbuff->buff[*n*]	Description
0	RFCOMM session number.
1	RFCOMM port number.
2	MSC byte.

Description

Indication of RFCOMM MSC command.

7.3.25 RFCA_UP_MSC_CNF – MSC Command Confirmation

Event Data

Pointer to Mbuff.

Mbuff->buff[*n*]	Description
0	RFCOMM session number.
1	RFCOMM port number.

Description

Confirmation of RFCOMM MSC command.

7.3.26 RFCA_UP_PN_IND – PN Command Indication

Event Data

Pointer to Mbuff.

Mbuff->buff[*n*]	Description
0	RFCOMM session number.
1	RFCOMM port number.
2	Priority
3,4	RFCOMM maximum frame size

Description

Indication of RFCOMM PN command. RFCOMM will reply automatically.

7.3.27 RFCA_UP_PN_CNF – PN Command Confirmation

Event Data

Pointer to Mbuff.

Mbuff->buff[*n*]	Description
0	RFCOMM session number.
1	RFCOMM port number.
2	Priority
3,4	RFCOMM maximum frame size

Description

Confirmation of RFCOMM PN command.

7.3.28 RFCA_UP_RLS_IND – RLS Command Indication

Event Data

Pointer to Mbuff.

Mbuff->buff[*n*]	Description
0	RFCOMM session number.
1	RFCOMM port number.
2	RLS byte

Description

Indication of RFCOMM RLS command. RFCOMM will reply automatically.

7.3.29 RFCA_UP_RLS_CNF – RLS Command Confirmation

Event Data

Pointer to Mbuff.

Mbuff->buff[*n*]	Description
0	RFCOMM session number.
1	RFCOMM port number.

Description

Confirmation of RFCOMM RLS command.

7.3.30 RFCA_UP_RPN_IND – RPN Command Indication

Event Data

Pointer to Mbuff.

Mbuff->buff[*n*]	Description
0	RFCOMM session number.
1	RFCOMM port number.
2	Baud rate (if length >2)
3	Port value(if length >2)
4	Flow control byte(if length >2)
5	XON character(if length >2)
6	XOFF character(if length >2)

Description

Indication of RFCOMM RPN command. RFCOMM will reply automatically.

7.3.31 RFCA_UP_RPN_CNF – RPN Command Confirmation

Event Data

Pointer to Mbuff.

Mbuff->buff[*n*]	Description
0	RFCOMM session number.
1	RFCOMM port number.

Description

Confirmation of RFCOMM RPN command.

7.3.32 RFCA_UP_START_IND – Session Up Indication

Event Data

Pointer to Mbuff.

Mbuff->buff[*n*]	Description
0	RFCOMM session number.
1	RFCOMM port number.

Description

Indication of RFCOMM session is up.

7.3.33 RFCA_UP_START_CNF – Session Up Confirmation

Event Data

Pointer to Mbuff.

Mbuff->buff[*n*]	Description
0	RFCOMM session number.
1	RFCOMM port number.

Description

Confirmation of RFCOMM session is up successfully.

7.3.34 RFCA_UP_START_CNF_NEG – Session Negative Confirm

Event Data

Pointer to Mbuff.

Mbuff->buff[*n*]	Description
0	RFCOMM session number.
1	RFCOMM port number.

Description

Confirmation that an attempt to open a RCOMM session failed.

7.3.35 RFCA_UP_TEST_CNF – TEST Command Confirmation

Event Data

Pointer to Mbuff.

Mbuff->buff[*n*]	Description
0	RFCOMM session number.
1	RFCOMM port number.

Description

Confirmation of RCOMM TEST command.

7.3.36 SDPA_UP_CLOSE_SUCCESS – SDP session is closed

Event Data

None.

Description

This event indicates the SDP session is closed as a result of calling the API `SDP_CloseChannel()`. This event is sent as soon as the SDP initiates the closing procedure even before the SDP Channel is actually closed.

7.3.37 SDPA_UP_DISCONNECT – SDP Session is disconnected

Event Data

None.

Description

This event is sent if the ACL Link goes off or if the remote side closes the SDP Channel.

7.3.38 SDPA_UP_ERROR_IN_REQ – SDP Error in the request

Event Data

None.

Description

This event may be sent when any of the API's `SDP_SerChannel()`, `SDP_SSAReq()`, `SDP_SSReq()`, or `SDP_SAReq()` are called. This event indicates that a SDP request is not correct. This event is sent if the remote side sends an error reply to a request or if the request frame size is more than `SDP MTUSIZE. SDP MTUSIZE` is 48.

7.3.39 SDPA_UP_ERROR_IN_RSP – SDP Error in response

Event Data

None.

Description

This event may be sent when any of these API's `SDP_SerChannel()`, `SDP_SSAReq()`, `SDP_SSReq()`, or `SDP_SAReq()` are called. This event indicates the SDP Client was not able to process a response received from the SDP server because either the server sent the wrong data or the structure given by the client application when calling the API was not sufficient to hold all the return values.

7.3.40 SDPA_UP_OPEN_FAILURE – SDP session is not established

Event Data

None.

Description

This event may be received as a result of calling the `SDP_OpenChannel()` API. This event indicates SDP session was not established with the remote side because either the remote side did not respond to this request or the remote side sent a negative reply for establishing a channel.

7.3.41 SDPA_UP_OPEN_SUCCESS – SDP session is established

Event Data

None.

Description

This event may be received as a result of calling the `SDP_OpenChannel()` API. This event indicates SDP session was established with the remote side.

7.3.42 SDPA_UP_RESPONSE – SDP Error in the request

Event Data

None.

Description

This event may be sent when any of the API's `SDP_SerChannel()`, `SDP_SSAReq()`, `SDP_SSReq()`, or `SDP_SAReq()` are called. This event indicates that a SDP request was completed and the result can be taken from the data structure passed as a parameter when calling the API.

7.3.43 USB Status Change Event

Event Data

UINT16. The status change message request is registered with the driver by calling **USBIoctl()** with a status change type code (see Section 7.2.72). The caller specifies the actual message value when the request is made.

Description

The mbuff parameter in the message contains the encoded status change. The status change message is sent by the USB driver whenever a suspend, resume, or reset condition is detected on the bus. Which condition has actually occurred is determined by examining the parameter supplied to the status change message handler.

The following status change codes are defined in **USBGref.h**:

Status Change Code	Meaning
USB_SCReset	Reset Detected. The host has issued a reset.
USB_SCResetEnd	Reset End Detected. The reset condition has ended.
USB_SCSuspend	Suspend Detected. At least 3 ms. of idle has been detected on USB.
USB_SCResume	Resume Detected. Host has started resume signaling.
USB_SCAltSettings	Alternate setting change. Host has changed packet size on isochronous endpoints.
USB_SCSetFeature	Set Feature received from host.
USB_SCClrFeature	Clear Feature received from host.
USB_SCDisconnect	USB cable electrically disconnected.
USB_SCEnumStarted	Enumeration started.

8 Files

This section describes various files that come with the product and the way each file is used.

8.1 Header Files

This section identifies the header files used to describe the function prototypes which provide the various services provided to the application. All header files are in **<install directory>\inc**.

Table 8-1: Header Files

Name	Description
8051sio.h	Provides serial I/O driver API.
ag_hs_api.h	Provides application gateway and headset profile API's.
gap_api.h	Provides GAP API.
Pte.h	Master include file for PTE. Includes all sub header files.
Pte_core.h	Header file for PTE "core" services. Included from **Pte.h.**
Pte_mbuf.h	Header file for PTE "mbuff" services. Included from Pte.h.
Pte_tmr.h	Header file for PTE "timer" services. Included from **Pte.h.**
rfc_api.h	Provides RFCOMM API.
sdp_api.h	Provides SDP API.
target.h	Contains XDATA memory mapped register declarations.
upperlayer_events.h	Contains AG, HS, RFCOMM, GAP, and SDP event message definitions.
USBGref.h	Provides USB driver API.

8.2 Project Files

This section describes each project file in the Embedded SDK. Each application directory contains a project file. Each project file causes a different configuration of hardware support and software capabilities to be built when the Keil compiler is invoked. Future versions of the SDK will have additional project files. The specific project files you can use depend on the version of the EDK you have. The following table describes each project file in the current version of the SDK, and the EDK(s) with which it can be used.

Table 8-2: Project Files

Name	Description	Use With EDK #
hscodec1.uv2	Builds Headset Profile (HSP) using the CODEC.	MT1750-EDK
hscodec2bs.uv2	Builds HSP using the CODEC and bank switching. Uses full 128K of flash memory.	MT1755-EDK
sppsio.uv2	Builds Serial Port Profile (SPP) over SIO 0.	MT1750-EDK, MT1760-EDK
sppusb.uv2	Builds SPP over USB. Requires host USB driver and virtual COM driver.	MT1760-EDK

8.3 Source Level Binding Files

This section describes the files used for source level binding. Source level binding is required to define threads and ISR's in the PTE environment. One copy of each source level binding file is in each application directory.

Table 8-3: Source Level Binding Files

Name	Description
PteConfig.c	Contains thread table and ISR table modifiable by user. Defines number of threads and number of data buffers.
PteConfig.h	Contains thread enumeration modifiable by user that must match thread table in **PteConfig.c**. Contains PTE configuration symbolic constants.

8.4 Source Level Configuration Files

This section identifies the files used for source level configuration. Source level configuration deals with such matters as defining the size of memory pools, defining the number of threads, and other configuration definitions which are determined at compile time. All source level configuration files are in **<install_directory>\src**.

Table 8-4: Source Level Configuration Files

Name	Description
lower.c	Contains **main().** Calls low level initialization routines.
PteApp.c	Source level PTE initialization. Not touched by user.
target.c	Defines XDATA memory mapped register assignments. Initializes serial I/O ports.

8.5 Library and Object Files

All library and object files supplied with the product are in **<install_directory>\lib**. These files are included as appropriate by the link stage of the build procedure and are combined with the object files produced by compiling the application source to produce ".hex" and ".omf" files which can be downloaded to a development board and executed.

9 Acknowledgements

Chris Alford authored a document on PTE operation from which large pieces have simply been cut and pasted into this document. Lily Guo generated the reference project files and associated descriptions. Manghai Thulasimani produced all the documentation on the SDP and GAP API's and events. Simei Du produced all the documentation on the RFCOMM and application gateway headset API's and events. Thomas Cook produced the PTE API and event documentation.

All errors and omissions are, of course, the author's responsibility.

10 References

[1] INTEL HEX FILE FORMAT, http://www.keil.com/support/docs/1584.htm

[2] KEEPING HEX RECORDS IN ORDER, http://www.keil.com/support/docs/1236.htm

[3] R80515 Microcontroller Specification, Appendix A – Instruction Set Details, Evatronix, <installation directory>\help\R80515_A.PDF

[4] R80515 Synthesizable HDL Core Specification, Evatronix, <installation directory >\help\R80515_SPEC.PDF

Index

S

T

Contact Information

World Headquarters

Microtune, Inc.
2201 Tenth Street
Plano, TX 75075 USA
Phone: 972-673-1600
Fax: 972-673-1602
Web: www.microtune.com
Sales: sales@microtune.com

Wireless Connectivity Division Offices

San Diego, CA

Microtune, Inc.
6440 Lusk Blvd, Suite D-205
San Diego, CA 92121 USA
Phone: 858-558-6088
Fax: 858-558-6598

Tokyo, Japan

Microtune, Inc.
Kawasho building 4F
3-10 Nihonbashi Kodemmacho
Chuo-ku, Tokyo 103-0001 Japan
Phone: +81 (0) 3-5652-0250
Fax: +81 (0) 3-5652-0251

Taipei, Taiwan

Microtune, Inc.
18F-7, No.77, Sec. 1, Hsin Tai Wu Rd.,
Hsi-Chih
Taipei Hsien, Taiwan
Phone: +886 (02) 2698-8648
Fax: +886 (02) 2698-8647

Singapore

Microtune, Inc.
1 International Business Park
#03-14A The Synergy
Podium Block, Singapore 609917
Phone: +65 567 9724
Fax: +65 567 9714