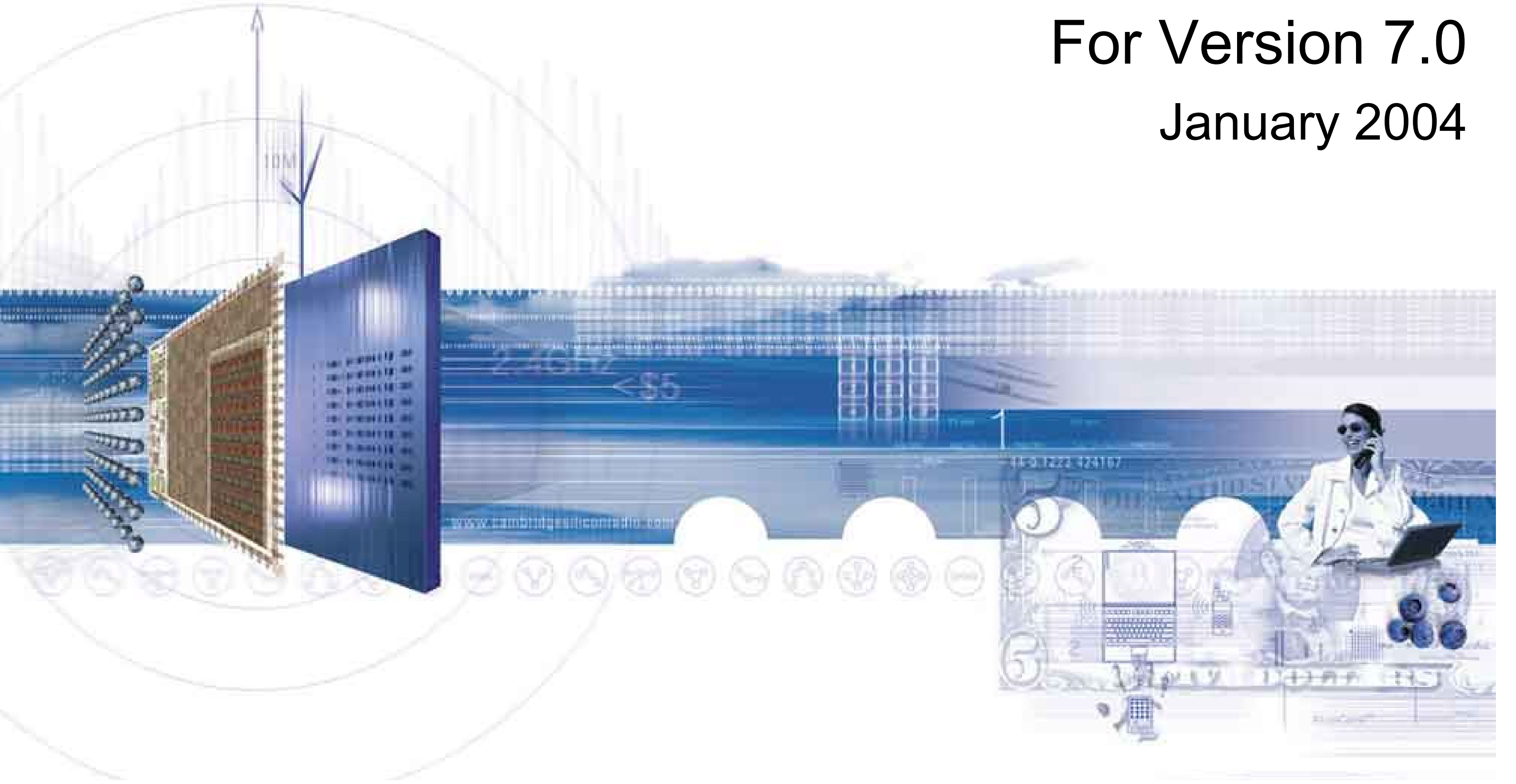
BlueCore™ Host Software

Porting Guide

BCHS GU 002

For Version 7.0

January 2004



CSR

Unit 400 Cambridge Science Park
Milton Road
Cambridge
CB4 0WH
United Kingdom

Registered in England 3665875

Contents

List of Figures

General Information

Ownership

All information contained in this document is owned by Cambridge Silicon Radio Ltd. and should not be copied for any purpose.

The BlueCore™ Host Software is owned by Cambridge Silicon Radio Ltd (CSR). The right to use this software is described in a separate license agreement.

Trademarks

Bluetooth® and the Bluetooth® logos are trademarks owned by Bluetooth® SIG Inc., USA and licensed to CSR.

BlueCore™ is a trademark of Cambridge Silicon Radio Ltd.

All other product, service and company names are trademarks, registered trademarks, or service marks of their respective owners.

Confidentiality

This document contains confidential information that is proprietary to CSR. This information must only be used for its intended purpose and should not be disclosed to third parties.

Liability

CSR's products are not authorised for use in life-support or safety-critical applications.

1 Introduction

1.1 General Information

BlueCore™ Host Software (BCHS) is developed to work with CSR's family of BlueCore™ IC's.

BCHS is intended for embedded products having a host processor for running the BCHS and the Bluetooth® application. BCHS together with the BlueCore™ IC is a complete Bluetooth® system solution from RF to profiles. BCHS includes much of the Bluetooth® intelligence and gives the user a simple API. This makes it possible to develop a Bluetooth® product without in-depth Bluetooth® knowledge.

Please read the license agreements [LIC] before use.

1.2 Audience

This porting guide gives an overview of how the BCHS can be ported to a platform defined by the end user and which tasks are involved in the porting. This document is a supplement to the BCHS-API-001_UserGuide and must be read in conjunction with the user guide.

The information contained in this porting guide is intended for system architects and system designers designing a system where BCHS must be included as a system component. Besides having in-depth knowledge of C-programming and real-time programming, the following competences are necessary for a successful porting of BCHS:

- Target OS/kernel (especially UART communication) to port to and how it is used by current applications
- The ported scheduler

It is also desirable to have basic knowledge of:

- Memory management principles
- Task scheduling and general kernel principles

1.3 Reading Information

The focus of this document is to describe how BCHS can be ported. The document is divided into chapters for the different areas that need consideration during porting, see also chapter 2. For detailed information about the various profile APIs, please refer to the API documentation for the relevant profile.

Information about the Scheduler API and how to port the Scheduler can be found in chapter 3.

A description of the function to access the persistent memory can be found in chapter 4.

An overview of YABCSP and how to interface the lower layers to the communication driver can be found in chapter 5.

In chapter 6 a description of the upper layer interface is given.

In chapter 7 it is describe how to configure the Audio over PCM or BCSP.

A porting example based on the Nucleus PLUS Kernel is described in Appendix A. The appendix is a user guide in porting the Scheduler API by implementing a new Scheduler, and how to port the UART driver.

2 Porting Model

With the BCHS package everything required for easy development of Bluetooth® applications is supplied. A central point regarding BCHS is that it is designed to be portable. BCHS is therefore designed with minimal dependencies on external systems and this again implies that BCHS executes with a very limited set of external functions.

Figure 1 illustrates the main components in a system with BCHS integrated and the external dependencies of BCHS. Please note that only a subset of the supplied Profile Managers is included in the figure.

The figure illustrates how the BCHS components execute with a Scheduler or a Scheduler API. Apart from the Scheduler API arrows represent the external interfaces. The number of external interfaces is kept to a minimum in order to ease integration into other subsystems. The external interfaces, which need to be ported, can be divided into:

1. **Scheduler API**, which is further described in [API].
2. **Persistent Memory Access** for storing of link keys of bonded devices symbolised by the Device DB.
3. **Lower layer UART interface** for BlueCore™ communication.
4. **Upper layer interface** symbolised by the APP2BCHS communication box.

A description of the process of porting the four interfaces is giving in the following chapters. It is recommended that the porting will be done in the same order. The porting affects the identified interfaces and depends on the existing host kernel to which the port is to be carried out. As a supplement to the following chapters, please refer to Appendix A, which is a user guide in porting to the Nucleus PLUS kernel.

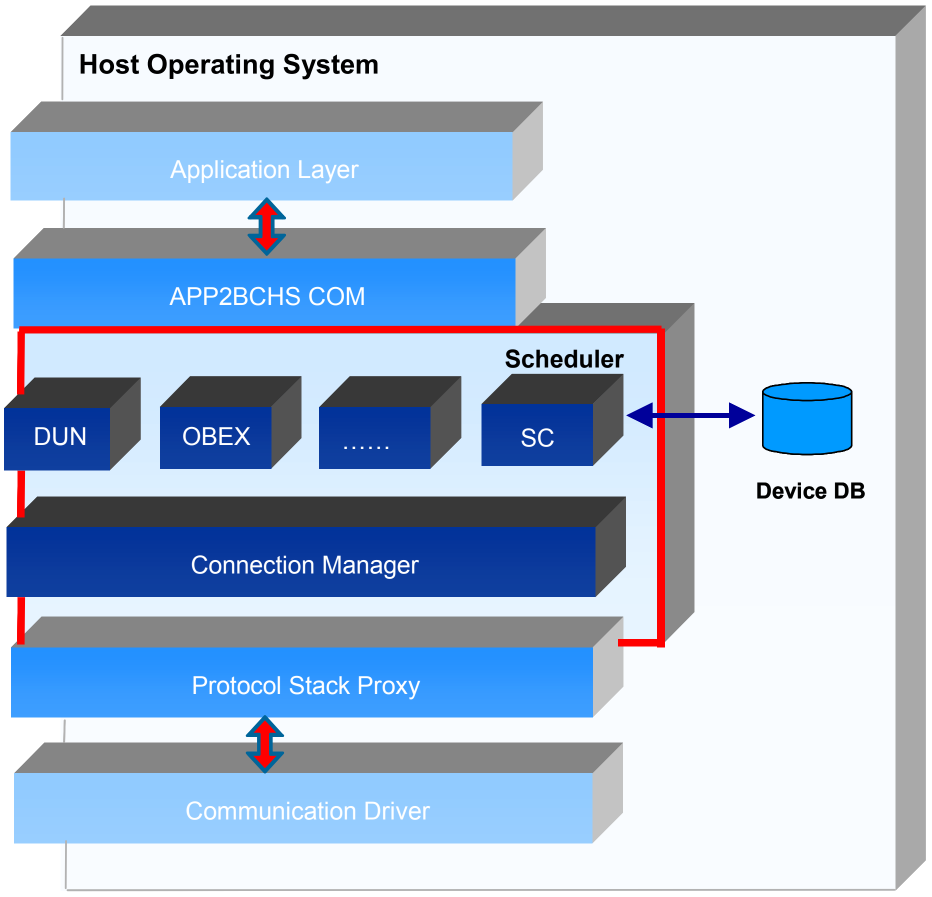


Figure 1: Porting Model

3 Scheduler API

Before starting porting the Scheduler API it is recommended that [API] is read thoroughly. This document outlines how the scheduler is organised, which makes it a fundamental part to understand.

3.1 Porting Method

Basically the porting of the Scheduler API can be done in two different ways:

1. By using the native kernels scheduling mechanism to provide the Scheduler API.
2. By implementing a dedicated Scheduler mechanism for BCHS.

It is also possible to use a mix of method 1 and 2. Which method to use depends on the actual host kernel to port to. If the host kernel provides functions where porting is straightforward or provides functions very similar to the ones required in [API], obviously it is preferable to use method 1. If it is not possible to use method 1, in full or part, method 2 can be used. In this method a Scheduler layer is implemented on top of the existing environment, which may be the host kernel. This adds a little overhead in terms of processing power required since the functions must be mapped to an existing kernel API. However, since the Scheduler API defined in [API] is very limited this is a simple task and the overhead is very limited. Whichever method is used, the key point is that the porting requires no changes in BCHS, but providing the functions defined by the interfaces only. The porting job is very much influenced by the choice of the porting method.

In Figure 2 a conceptual model of the encapsulation of the BCHS layers is illustrated. Depending on the actual porting method, the Scheduler may be an actual component layer in BCHS or it may merely be a virtual layer providing the API layer only.

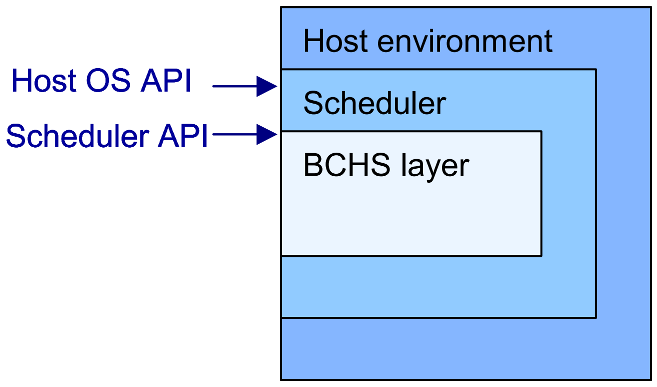


Figure 2: BCHS Bluetooth® functional layer to Scheduler encapsulation

3.2 Scheduler Porting

This section is a general guide of how to port the Scheduler to a native kernel according to method 2, see section 3.1. The features mentioned must be considered and ported to the kernel in which the Scheduler is executing.

Please note that when the Schedule API is ported, test cases to validate the port of this API can be found in *Drivers/Scheduler/Test* (path relative to installation path).

3.2.1 Controlling the Scheduler from the Native Kernel

The Scheduler examples provided with BCHS are initialised with a call to a function called *init_sched*. This function must be called from the host application and will initialise the necessary internal Scheduler variables. This is also the function in which any other initialising functions can be located, e.g. initialise memory if needed.

Once initialised, the Scheduler examples are started with a call to the function *sched*. The *sched* function contains the Scheduler functionality and will not return (because it enters an endless loop, which schedules the BCHS tasks according to the round-robin principle).

3.2.2 Task Scheduling

As described in [API], the scheduler organises the code as a set of tasks, and provides means for communication between the tasks. Every task requires one message queue only. All tasks in BCHS are defined to be equal regarding run level priority, and are **not** designed to be re-entrant. Hence BCHS tasks do not make any assumption on task priority or execution sequence; all tasks are defined with one run level.

The tasks defined in the BCHS Scheduler examples are all defined in the *tasks.h/c* file. The *tasks.c* file contains the queue identifiers for the different tasks. Tasks that are added as a result of the port and visible to the scheduler must be added to the *tasks.h/c* file. These are also the tasks for which an initialization and task handler function is defined. If the initialization function is NULL defined it is not used (see *tasks.c*).

BCHS is supplied with multiple profiles. Depending on the application it may be feasible to include/exclude one or more of the profile layers. To exclude a profile layer, use the:

- EXCLUDE_XXX_MODULE

define. All modules is defined in the BCHS User Guide, see BCHS-GU-001_UserGuide.

During start-up, i.e. in the initialisation function, a task may use the Scheduler API to communicate with other BCHS tasks. It is the responsibility of the Scheduler/host OS to ensure that BCHS tasks are not actually invoked before the initialisation function is completed for all defined and included components.

When the task handler is called, the BCHS task will complete the function related to the event received; the function will run until completion and not hand over control to the calling function until this point. All BCHS tasks are designed to complete as soon as possible.

3.2.3 Background Interrupt

Since the Scheduler is not re-entrant safe, foreground tasks, such as UART communication drivers, must not interrupt the Scheduler background job. Instead a mechanism to inform the background task that a foreground task needs attention must be implemented.

In the Scheduler examples are provided by BCHS and the functions that need consideration for background interrupt handling are:

- register_bg_int
- bg_intx, where x is a number indicating the interrupt number

Background interrupts are expected to be used to allow a foreground task to tell the background task (the Scheduler) that data is sitting in a buffer, and is available for analysis by the background task. This will avoid that the background task needs to poll the buffers. The foreground can call function *bg_intx* with x being the requested background interrupt number.

A background interrupt is registered with a call to *register_bg_int.* This function must be called from within the function that makes use of the background interrupt function. The *register_bg_int* function defines the interrupt number and the function to be activated. In section 5.1 it is described how the UART drivers used this mechanism.

Please note that example code of register_bg_int and *bg_intx* functions can be found in *the bg_int.c/.h* files.

3.2.4 Message Forwarding

Every task requires one message queue only. It is the responsibility of the host kernel or the Scheduler to handle messages passing between the task, both internally between BCHS tasks as well as between BCHS task and external components. The functions that need to be considered when porting message forwarding are:

- put_message
- cancel_message
- get_message
- put_message_in
- put_message_at
- cancel_timed_message

Please refer to [API] for a detailed description of these functions.

Please note that a task's message handler is allowed to forward/cancel a message to any task's queues, but it is only allowed to consume messages from its own queue.

If the scheduler is defined as a task in the host kernel, and the host kennel wants to send a message to a task in the Scheduler, the host kernel is responsible for sending this message to the Scheduler. Likewise, sending a message from a task within the Scheduler, the message must be transmitted to the task in the native kernel by the Scheduler.

3.2.5 Timer Handling

BCHS makes use of timed events. This implies that a timer function needs to be ported such that the timer functions used by the code are available.

The functions that need to be considered when porting are:

- timed_event_in
- timed_event_at
- cancel_timed_event
- get_time

Please refer to [API] for a detailed description of these functions.

3.2.6 Dynamic Memory Allocation Handling

BCHS makes use of dynamic memory allocation and de-allocation. The functions that need to be considered when porting dynamic memory allocation are:

- pmalloc
- zpmalloc
- pfree

Please refer to [API] for a detailed description of these functions.

For simple targets (targets without any kernel) the *pmalloc* function can be ported to run directly on the hardware without interfacing to a kernel. The *pmalloc* call is analogous to the *malloc* call defined in ANSI C – it asks for a memory block of the indicated size. If the memory request can be successfully met, a pointer to the allocated memory block is returned. If the request cannot be met the Scheduler or host kernel must ensure proper action, e.g. restart the system.

Depending on the environment to which BCHS is ported, memory management can be implemented in different ways. One way is to reuse the *malloc* call if such exists; in this case the porting is simply a matter of calling the *malloc* function from within the *pmalloc* function. It must however be considered if memory allocation, fragmentation etc. is sufficiently efficiently implemented for real time systems. If this is not the case another way is then to use pre-allocated memory with memory allocated from a set of pools. If pre-allocated memory is used for *pmalloc* it must be assured that sufficient memory is allocated in the pools. The major loss using pool base memory is the inability to handle requests for arbitrary-sized memory blocks and an inevitable overhead of memory allocated.

Memory claimed with *pmalloc* must be freed with a call to *pfree. pfree* must return the previously allocated memory to the system for reuse. Again, if *pmalloc* is implemented via a call to *malloc* porting the *pfree* is a very simple task of simply mapping the *pfree* function to standard *free*.

The only difference between the *pmalloc* - and the z*pmalloc* function is that z*pmalloc* also sets the obtained memory bytes to zeroes.

3.2.7 Exception Handling

Please note that the BCHS component does not implement any exception handling for abnormal situations (except for Bluetooth® protocol exception handling). This implies that any exception handling must be part of the Scheduler or host kernel, e.g. memory exhaustion must be properly handled by the Scheduler or the host kernel.

The action to take when an exception arises must be determined in the Scheduler or in the host kernel. Typical actions may be to rewind the system to a known secure position or simply restart the system.

In the Scheduler examples any serious exception that may arise will initiate a call to the *panic* function, which can be found in the *panic.c* file. Situations that may lead to a panic function call are defined in the *panic.c* file. Examples are if the system runs out of memory, trying to send a message to an unknown task id or too many messages are put on queue.

In BCHS the panic function will only be called from:

- YABCSP (in *txmsg.c*), if a received message is larger than its allowed maximum
- *HcCom.c,* if it receivs an unknown primitive or signal, from one of the Scheduler tasks. Please note that *Hccom.c* is only called in the RFCOMM build
- *buffer.c*, if there is some buffer congestion. Please note that *buffer.c* is only called if the HCI build is used

3.2.8 Controlled Shut Down

BCHS includes an optional possibility to close down all tasks in a graceful manner; i.e. all allocated resources are de-allocated and all messages on the tasks input queue are released. The feature is optional and includes using the compiler switch *ENABLE_SHUTDOWN*. In environments where the target has full control of all resources or in OS systems where BCHS runs as a task in that OS, and the OS releases all task resources when the task is terminated, it may not be necessary to implement the close down feature. In case BCHS needs to be re-initialised due to e.g. a BC chip reset, it is necessary to do a BCHS shut down followed by an initialisation of BCHS. It is not enough just initialise BCHS again as this may result in memory leaks.

If the *ENABLE_SHUTDOWN* switch is defined all profile managers will include a de-initialisation function. If other tasks are added during the porting these may also have to define a de-initialisation function. The scheduler can call the de-initialisation functions during close down. The de-initialisation functions are defined in the *tasks.c* file; any additional tasks that may have been added during the porting also need to be defined in the tasks files. In the porting examples provided with BCHS it is necessary to shut down BCHS using the de-initialisation functions to prevent memory leaks. Please note that only the Windows and ARM stand-alone scheduler examples include the shut down feature.

Before calling any of the de-initialisation functions the scheduler must:

- ensure that no more messages are put on the task input queues
- move all timed messages (from the *put_message_in* and *put_message_at functions*) to the tasks input queue

The scheduler should then call all the task de-initialisation functions in sequence. The tasks will empty the input queue and cancel timed events (from the *timed_event*_in and *timed_event*_at functions) and allocated data. If the tasks do not cancel all timed events it is up to the scheduler to cancel and de-allocate the remaining timed events. The scheduler must also assure that any data pointed to by the *mv* pointer (in the *timed_event_in* and *timed_event_at* functions) is de-allocated using the *pfree()* function.

An example showing the use of shut down can be found in the Windows or ARM audio gateway demo.

The functionality needed in the application layer after the scheduler loop is terminated is:

```
if shut down is activated
{
        •  Exit scheduler loop, sched_stop()
        •  Stop the uart driver, UartDrv_Stop()
        •  Remove any messages in the abcsp layer calling abcsp_deinit()
        •  Transfer timed messages to the task input queue, let the scheduler call the task de-
           initialisation functions and remove any remaining timed events and de-allocate any data
           associated with the timed event, sched_task_deinit()
}
```

Note that the close down event is manually generated by the user and handing of the user event is included in the *agdemoapp*.c file.

4 Persistent Memory Access

BCHS includes link key management for bonded devices. In order to preserve the link keys when devices are powered off, BCHS must have access to persistent memory where the link keys can be stored and maintained. The security manager in BCHS calls the function:

- scDbRead
- scDbWrite

which can be found in the *sc_db.c* file. For a more thorough description of how to port these functions, please refer to the BCHS Security API, see BCHS-API-003_SecurityApi.pdf.

The *scDbRead* function is used to read the link key of a stored device. If the specified device does not exist, a NULL pointer is returned. To store a key of a new device or replace/update an exiting key, the function *scDbWrite* is used.

It must be noted that BCHS can execute without access to persistent storage but with a simple function stub for the read and write function. In this case of course it is not possible to retain the link keys in between sessions where the system has been reset, e.g. power off.

5 Lower Layer UART Interface

Since the Bluetooth® protocol stack is divided into a part residing on host (BCHS) and a part residing on the BlueCore (Bluetooth® Core stack), a communication media and protocol between the processes running on host and in the BlueCore is required. The communication protocol provided by BCHS is a version of the BCSP protocol called YABCSP. Detailed information about YABCSP can be found in [YABCSP].

BCSP defines logical channels, which can be used for routing of signals between the host software and the software in the BlueCore. In the RFCOMM build model, all Bluetooth® protocol stack layers up to and including RFCOMM and SDP reside in the BlueCore. Only the Profile Managers (and thus also the Scheduler) must be executed in the host processor. In the HCI build model the Bluetooth® Core stack is located on the host platform and is therefore the proxy interface upper layer.

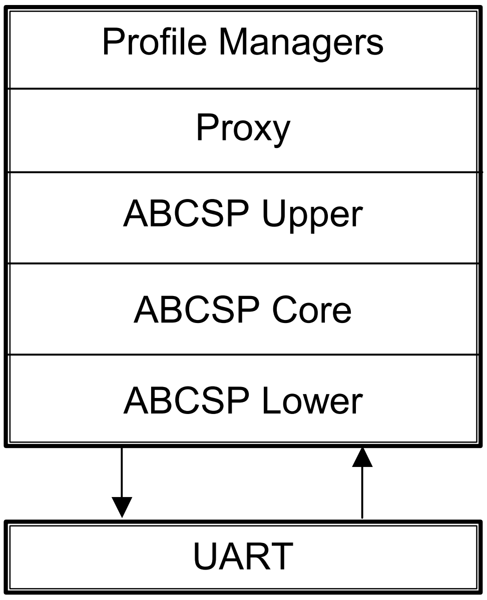


Figure 3: RFCOMM build model

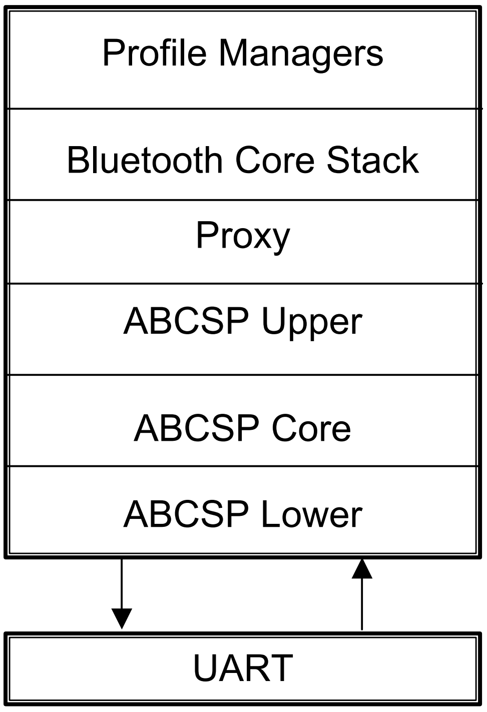


Figure 4: HCI build model

In the following section it will be described which part of the lower layers that must be considered when the Scheduler API is ported by implementing a dedicated Scheduler mechanism for BCHS, which is the case for the examples provided with BCHS.

5.1 Porting the UART Interface

If the Scheduler API is ported, by implementing a dedicated Scheduler mechanism for BCHS, the porting job of YABCSP is simplified, because then only the lower layer, which defines the UART interface, and the initialization function for YABCSP, must be considered. The functions that must be considered is listed below:

- abcsp_init
- UartDrv_Configure
- UartDrv_RegisterHandlers
- UartDrv_Start
- UartDrv_Rx
- UartDrv_Tx

5.1.1 abcsp_init

The *abcsp_init* function is used to initialise the YABCSP library. The *abcsp_init* function must be called from the main function before the Scheduler initialisation function, because *abcsp_init* generates some background interrupt requests that otherwise could be lost, and before the blocking Scheduler function.

5.1.2 UartDrv_Configure

The *UartDrv_Configure* function must be implemented by the developer and must be used to initialise the UART driver. An example of how this can be done, for the Windows platform can be found in the *SerialCom.c* file.

5.1.3 UartDrv_RegisterHandlers

In the porting examples provided with BCHS, the *UartDrv_RegisterHandlers* function is used to register the background interrupts used for the UART communication driver, and is called from the main function. The UART driver must use:

- Background interrupt number 1, to inform the Scheduler that data is available in the UART RX buffer. This must be done, by calling the function *bg_int1*
- Background interrupt number 2, to inform the Scheduler that UART data needs to be sent to the UART TX buffer or to check if an ACK needs to be send. This must be done, by calling the function *bg_int2*

These two background interrupt must be registered by the *UartDrv_RegisterHandlers* function by calling:

- register_bg_int(1, UartDrv_Rx)
- register_bg_int(2, abcsp_pumptxmsgsOut)

Please note that the processing of data internally in the YABCSP is controlled by the *abcsp_pumptxmsgsOut.* The *bg_int2* interrupt is issued by the function called *abcsp_req_pumptxmsgs* from the YABCSP. The *abcsp_req_pumptxmsgs* must not call the *abcsp_pumptxmsgsOut* directly. The *abcsp_req_pumptxmsgs* is called automatically when the *UartDrv_Tx* is called or *UartDrv_RX* is called. This is because the YABCSP library needs to make some internal checks and prioritise the messages that should be sent.

If the ROM version of the chip is used, the UART driver must also use:

- Background interrupt number 6, to inform the Scheduler that YABCSP needs to be restarted. This must be done, by calling the function *bg_int6*

This background interrupt must be registered by the *UartDrv_RegisterHandlers* function by calling:

- register_bg_int(6, abcsp_restart);

5.1.4 UartDrv_Start

The *UartDrv_Start* function is only strictly necessary if the ROM version of the chip is used and it needs to change the baud rate doing initialisation. The function must be implemented by the developer and must be used to start/restart the UART driver. An example of how this can be done, for the Windows platform can be found in the *SerialCom.c* file, and in the bccmdBootStrap.c file. An example is given of how this function is used to restart the UART driver with another baud rate.

5.1.5 UartDrv_RX

The YABCSP is not re-entered, so when the UART driver receives a message from the Bluetooth Core stack it must inform the Scheduler by calling the *bg_int1* function. When the Scheduler is informed it calls the *UartDrv_RX* function, see section 5.1.2.

UartDrv_RX must take the available bytes out from the UART RX buffer and push them to the YABCSP library by calling the function *abcsp_uart_deliverbytes*. The function *abcsp_uart_deliverbytes* is defined in the *abcsp.h* file, and returns the number of consumed bytes.

If bytes are still available in the UART RX buffer after *abcsp_uart_deliverbytes* is called, the *UartDrv_RX* function must inform the Scheduler by calling the *bg_int1* function.

5.1.6 UartDrv_TX

The *UartDrv_TX* is used to push a message onto the UART TX buffer. *UartDrv_TX* is called from the *ABCSP_UART_SENDBYTES* macro (see the *config_txmsg.h* file) and is defined as:

*bool_t UartDrv_Tx(char *buf, uint16_t num_to_send, uint16_t *num_send)*

UartDrv_TX must push *num_to_send* bytes from *buf* to the UART TX buffer. If the UART TX buffer is full it must return FALSE and *num_send* must be 0. If the bytes are pushed to the TX buffer the function must return TRUE.

5.2 Proxy Related Parameter Configuration

In the RFCOMM build the number of messages that can be sent to BlueCore must be controlled in order not to flood the chip with messages. The BCHS file *HcCom.h* have defines for:

1. *TX CREDIT ISSUE THRESHOLD*, which is how often the BlueCore will issue a credit update to the host. I.e. when the number of primitives sent to the BlueCore passes this threshold, a credit update will be issued to host indicating the number of consumed primitives.
2. *TX CREDIT ISSUE TIMER* defines the time between checks that the number of unconsumed primitives in the BlueCore has changed. If the number has changed, a credit update will be issued to the host indicating the number of consumed primitives since last credit update.

A credit update will be issued when one of the conditions are fulfilled, which ever appears first. The *rfc_build_proxy.h* file contains an algorithm for automatic calculation of these parameters. If, however it is necessary to adjust these parameters it can be done simply by changing the mentioned defines. The reason for adjusting the proxy credit setting is that this may influence the overall data throughput. Please note that setting the *TX CREDIT ISSUE TIMER* to 0 disables the timer.

The purpose of the proxy for the HCI build is two fold:

1. ensure identical interfaces as for RFCOMM build.
2. map message data to/from the message block structure, i.e. allow use of segmented data in the Bluetooth Core stack.

6 Upper Layer Interface

Usually the Scheduler will run as a separate process in the host kernel. If other processes are defined and running in the native kernel and they need to communicate with any component running in the Scheduler, clearly an interface between the application layer (running in the native kernel) and the Scheduler must be made. In a porting according to porting method 1 (see section 3.1) this may not be required. It is possible to implement the application layer to run in the Scheduler as well, but this is normally not the case, as BCHS typically is integrated into an existing product where the application layer must be reused.

The upper layer interface (APP2BCHS in Figure 1) is responsible for communication between tasks executing in the host kernel and tasks executing in the Scheduler. Consequently, this interface is only needed if the Scheduler API is ported, by implementing a dedicated Scheduler mechanism for BCHS, and if the application is executed in the host environment. Hereby downstream and upstream messages must be treated as described in the following subsections.

6.1 Downstream Messages (Host to Scheduler)

When messages are sent from an application task in the host kernel destined to a task in the Scheduler, APP2BCHS must intercept these messages and inform the Scheduler that an incoming message is waiting. The incoming message can be retrieved from the Scheduler dedicated call in the *sched* function or through use of the background interrupt mechanism.

The function used to retrieve the message from the host kernel must ensure that the format for the messages in the Scheduler complies with the format defined in the Profile Manager API documents. As mentioned in 3.2.2 each Scheduler task has its own message queue. The retrieved message can be put on the Scheduler task queue using the *put_message* function call. The *put_message* function has a parameter, which identifies the task queue on which the message must be stored (see [API]). The queue identifier can be included in the host message or a conversion may take place in APP2BCHS.

6.2 Upstream Messages (Scheduler to Host)

Messages sent from a Scheduler task to a host environment task are identified based upon the destination queue identifier. When the Scheduler *put_message* identifies a message for this particular queue, the message is intercepted and routed to the host kernel. Another possibility is to let each host task be represented in APP2BCHS with a pseudo task handler for these tasks. The Scheduler will call the task handler function when a message is stored on the queue. The task handler must then consume the Scheduler message and pass it on to the right host kernel task using the host kernel message send function.

Depending on the host environment it may be possible to map the Scheduler message directly using the queue identifier. If this is not possible the message must be mapped in APP2BCHS.

7 Audio Configuration

SCO data can either be routed over the PCM port or over BSCP.

7.1.1 Audio over PCM

In order to map SCO data over the PCM port set MAP_SCO_PCM PSKey to TRUE. Note that when SCO data is mapped over the PCM port, only one SCO connection is allowed.

7.1.2 Audio over BCSP

In order to send and receive audio over BCSP the MAP_SCO_PCM PSKey must be set to FALSE and the application must register a SCO service function. The SCO service function must be registered when the application receives an audio confirm/indication signal with result code SUCCESS and the SCO handle included.

The SCO service is registered by calling:

```
bool_t RegisterScoHandle(uint16_t theScoHandle, ScoHandlerFuncType theFunctionPtr)
```

which returns a TRUE if the call is successful. The `theScoHandle` is the identity of the SCO handle, which is returned as a parameter in the audio confirm/indication signal. The `theFunctionPtr` must be the name of the function, which the received HCI SCO data packets are sent to. This function must be defined as illustrated below:

```
void nameOfFunction(char * theData)
```

where `theData` is the HCI SCO data packet, see Figure 5. For more information about HCI SCO data packet refer to [BT11] part H:1, section 4.4.3.

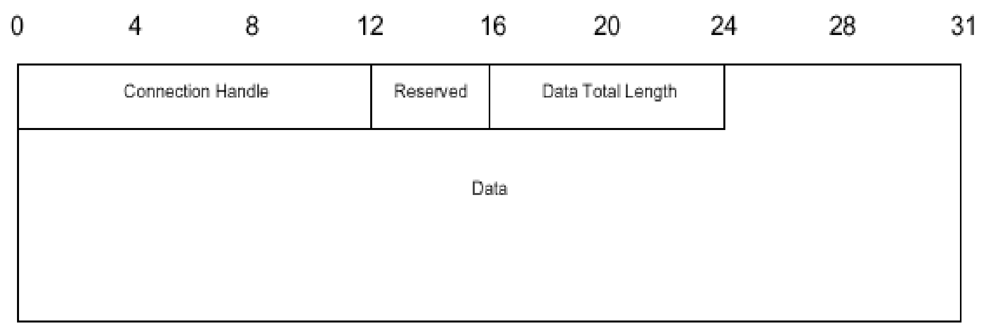


Figure 5: HCI SCO data packet

If the SCO service is registered with success, SCO data can be sent to the peer device by calling the function:

```
void SendScoData(char * theData)
```

The parameter `theData` must be a HCI SCO data packet, where the *Connection Handle* again is the identity of the SCO handle. *Data Total Length* is the length of SCO data giving in bytes, and *Data* is the SCO data. The SCO data format must be the same format as the incoming HCI SCO data packet.

The functions `registerScoHandle` and `SendScoData` are defined in AudioBCSP.h. In UsrConfig.h it is possible to set the voice parameter, which controls all the various settings for SCO connections. These settings apply to all SCO connections, and cannot be set individually.

NOTE: when audio is transferred as a 16 bit data stream, CSR recommends a minimum of 460.8 kb/s. For 2 or 3 simultaneous SCO links, CSR recommends a minimum of 921.6 kb/s. For 8 bit data streams, the baud rate requirements can be halved.

Appendix A: Porting Example for Nucleus PLUS

A1. Introduction

BlueCore™ Host Software (BCHS) is platform independent and highly portable. The main porting job in porting BCHS to a new platform is porting the virtual kernel API, the Scheduler API.

The porting example described in this appendix is based on the Nucleus PLUS which is a real-time preemptive, multitasking kernel designed for time-critical embedded applications. More information on Nucleus PLUS can be found at the http://www.acceleratedtechnology.com web site, and the Scheduler example files can be found in *Drivers/Scheduler/ARM/Nucleus* (path relative to installation path).

The information contained in this porting guide example is intended for system architects and system designers designing a system where BCHS must be included as a system component. Besides having in-depth knowledge of C-programming and real-time programming, knowledge of the Nucleus PLUS kernel system is required.

A2. Overview

A very central point with BCHS is that it is designed to be portable.

This document is a user guide in:

- Porting the Scheduler API by implementing a new Scheduler, which takes advantages of some of the functions offered by the Nucleus PLUS kernel
- Porting the UART interface for BlueCore™ communication to the Nucleus PLUS kernel, by using the implemented Scheduler

In this porting example it has been decided that the Scheduler will run as a separate task in the Nucleus PLUS kernel, because BCHS typically is integrated into an existing product where the application layer must be reused, as illustrated on Figure 6.

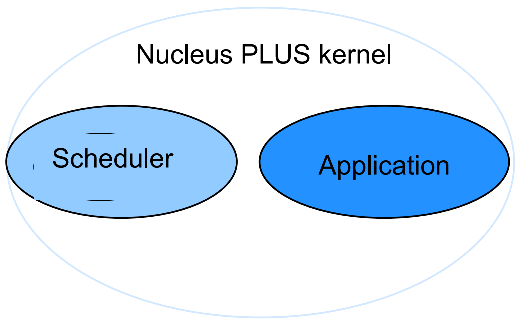


Figure 6: The Scheduler and the Application run as separate tasks in the Nucleus PLUS kernel

In order to communicate between the Nucleus tasks the Queue mechanism provided by the Nucleus PLUS kernel is used. It has been decided that each Nucleus task only has one dedicated Queue, e.g. if the Application task needs to send a message to the Scheduler task it has to put it on the queue dedicated to the Scheduler task, and vice versa. For more information about the Nucleus Queues, please refer to [NU].

The *Application_Initialize* function executed by Nucleus PLUS prior to starting the system is responsible for defining the initial application environment. In this case the *Application_Initialize* function creates a system memory pool, two tasks (respectively the Scheduler - and the Application task), two queues (one dedicated to the Scheduler - and the other to the Application task), and finally it creates a High-Level Interrupt Service Routine (HISR).

The *Application_Initialize* function will look like this:

```
void Application_Initialize(void *first_available_memory)
{
        Create a system memory pool
        Allocate the memory for the Scheduler task
        Create the Scheduler task
        Allocate memory for the Scheduler queue
        Create the Scheduler queue
        Allocate the memory for the Application task
        Create the Application task
        Allocate memory for the Application queue
        Create the Application queue
        Allocate the memory for the HISR control block
        Create the HISR
}
```

Please note that in Nucleus PLUS it is possible to create multiple memory pools (could be one for each task), but in order to minimize the overhead it has been decided to use a common system memory.

After the Nucleus *Application_Initialize* function has defined the application environment, it has to execute the nucleus task dedicated to the Scheduler and to the Application. In the following chapters an example is given of how these tasks can be implemented in order to port BCHS to the Nucleus PLUS kernel OS.

A3. The Scheduler

This chapter gives an example of how to port the Scheduler API to Nucleus PLUS kernel. Before reading the example it is recommended that [API] is read thoroughly, as it outlines a description of some of the Scheduler's interfaces.

Basically the porting of the Scheduler API can be done in two different ways: by using the native kernel's scheduling mechanism to provide the Scheduler API or by implementing a dedicated Scheduler mechanism for BCHS.

In this porting example the Scheduler API is provided by implementing a dedicated Scheduler. The Scheduler will take advantages of some of the Nucleus PLUS functions, defined in [NU].

A3.1 The Main Function

The main function of the Nucleus Scheduler task looks like this:

```
nucleusSchedTask(UNSIGNED argc, VOID *argv)
{
        UartDrv_Configure
        UartDrv_RegisterHandlers
        init_sched
        abcsp_init
        sched
}
```

The *UartDrv_Configure* and the *UartDrv_RegisterHandlers* functions are called to initialize and register some background interrupts for the UART driver. The serial protocol that BCHS provides is a version of the BCSP protocol called YABCSP. For more information of how to port these functions, refer to chapter A3.6.

Where *init_sched* function initializes the scheduler message queues, *abcsp_init* initializes ABCSP, and the *sched* function is in the primary code structure in the Scheduler.

Please notice that the *abcsp_init* function must be called after the *init_sched* function and before the *sched* function. The reason why it must be called after the *init_sched* function is because it generates some background interrupt requests that could be lost otherwise, and the reason why it must be called before the *sched* function is because it enters an endless loop and therefore never returns.

A3.2 The Scheduler Function

The primary code structure in the Scheduler is a set of tasks passing messages to each other. Each task has only one queue attached. The scheduler may place messages on any queue, but may only consume messages from its own queue.

The tasks defined in this example are all defined in the *task.h/c* file. These two files also contain the queue identifiers for the different tasks together with each tasks initialisation and handler functions. If a task's initialisation function is NULL defined, it is not used. BCHS is supplied with multiple profiles. Depending on the application it may be feasible to include/exclude one or more of the profile layers. To exclude a profile layer, use the EXCLUDE_XXX_MODULE define. For more information about which modules are defined, refer to the BCHS user guide document [USERG].

Please notice that in the *task.h* file, the application tasks queue is defined as an external task queue, because it has been decided that the application must run in its own Nucleus task, as explained in chapter A2.

The *sched* function contains the Scheduler functionality and will not return (because it enters an endless loop, which schedules the BCHS tasks according to the round-robin principle), after it has called the internal Scheduler tasks initialization functions. The basic functionality is:

```
call each Scheduler task's initialization function
endless loop
{
    if there is a message on one of the internal task queues
    {
        find a message on a task queue, and call its handler function
    }

    if there are NO messages on the internal Scheduler queues
    {
        Nucleus is allowed to suspend the Scheduler task if its Nucleus queue is empty.
    }
    else
    {
        Nucleus is NOT allowed to suspend its Nucleus queue if it is empty
    }

    if there is a message on the Nucleus queue
    {
        service it. (It can be a message from the application task, a timed event, or a background IRQ
    }
}
```

Please notice that the Scheduler is implemented as a blocking function, it is also possible to implement it as non-blocking.

The essential when designing the Scheduler loop as a blocking function, is that it only can be blocked one place in the loop. The purpose of this is that the Scheduler then can remain blocked until an external event occurs.

In this porting example, this has been achieved by making all external events, queue dependent. It is illustrated in the pseudo code that the Scheduler will be suspended/blocked if there are no messages on the internal Scheduler queues and there are no messages on its dedicated Nucleus queue. If the Scheduler is suspended it will be suspended until a message is available on its Nucleus queue, e.g. a message from the application task, a timed event or a background interrupt.

A3.3 Timer Handling

One of the major methods of invoking code under the Scheduler is to use timed events. A timed event is a function called after a particular system time. The functions that need to be considered when porting timed events are:

- timed_event_in
- timed_event_at
- cancel_timed_event
- get_time

Please refer to [API] for a detailed description of these functions, and for the example function, please refer to *sched.c* and *sched_private.h*.

A3.3.1 Design Overview of Timer Handling

This porting example takes advantage of the programmable timers that are provided by the Nucleus PLUS kernel. This decision was made because these timers can be created and deleted dynamically. There is no preset limit on the number of timers an application may have. Another feature with these timers is, that an Id is supplied to a user-supply High-Level interrupt routine. For a detailed description of the Nucleus PLUS timers, please refer to [NU].

In this design the user-supply interrupt routine is just sending a message, containing the timer Id and a message type, to the Scheduler's Nucleus queue. This is done in order to keep the Scheduler re-entrant save.

The following sections describe how the Scheduler timed events are designed. Basically what happens is that every time the Scheduler needs to use a timed event, it saves the giving parameters on an internal queue and create a new Nucleus PLUS timer. Each of these timers uses the same user-supply expiration routine, which is executed as a high-level interrupt service routine called *NucleusTimerExpirationRoutine*. This service routine builds a Nucleus PLUS message, containing the supplied timer Id, the message identifier defined as TIMEOUT_EVENT (which helps the Scheduler to identify that a timeout interrupt has occurred) and places it at the back of the Nucleus Queue, dedicated to the scheduler.

A timer Id is used in a message send to the Nucleus queue, which can be used to help the application to identify timers using the same expirations routine.

A3.3.2 The Scheduler Timed Event Queue

In order to keep track of the timed events the Scheduler internally holds a single timer link list. The timer link list is defined in *sched_private.h* as a structure called *TimedMessageType*, which consists of six elements:

- An unsigned integer *fniarg*, holding the *fniarg* parameter obtained from the timed_event_in or timed_event_at function.
- A void pointer *fnvarg*, holding the ** fnvarg* parameter obtained from the timed_event_in or timed_event_at function.
- A void pointer *(* eventFunction) (uint16_t, void *),* holding the function that the Scheduler must call when the timer has expired. This parameter is obtained from the timed_event_in or timed_event_at function.
- An unique timer identifier *id*, which is used to keep track of the timed event, or can be used to prevent a timed event.
- A TimedEventType pointer *next*, pointing at the next message on the list. If it is the last timed event on the list, the pointer value is NULL.
- A NU_TIMER pointer, holding the Nucleus PLUS user-supplied timer control block.

A3.3.3 timed_event_in

The *timed_event_in* function asks the Scheduler to call the function *fn(fniarg, fnvarg)* after at least period *delay* has passed. The function allocates memory for the new timer structure, the parameters in the structure are set, a new Nucleus PLUS timer is created, and the new timer structure is placed on the internal timer link list. The function returns the timer identifier *tid*, which can be used to prevent this timed event by calling the *cancel_timed_event* function, see section A3.3.5. The *timed_event_in* function looks like this:

```
tid timed_event_in(TIME delay, void (*fn) (uint16_t, void *), uint16_t fniarg, void *fnvarg)
{
    convert the given delay from microseconds to timer ticks
    call the function create_timer, which does the actual work.
    return the timer identifier tid obtain from the function create_timer
}
```

Please note that the *delay* is stated in microseconds and that the Nucleus PLUS timers use timer ticks. A tick is the basic unit of time in all Nucleus PLUS facilities, where each tick corresponds to a single hardware interrupt. The amount of actual time a tick represents is usually user-programmable.

As illustrated in the pseudo code, the *create_timer* function does the actual work of *timed_event_in,* and looks like this:

```
tid create_timer(UNSIGNED initialtime, void (*fn) (uint16_t, void *), uint16_t fniarg, void *fnvarg)
{
    allocate the memory for the new timer structure.
    obtain an unused timer identifier, tid
    try to create a new Nucleus PLUS timer and ask it to:
        call the function NucleusTimerExpirationRoutine after it expires,
        and supply the obtained timer identifier.

    if the Nucleus timer is created
    {
        set the parameter in the new timer structure and place it on the timer's internal timer link list
        return the timer identifier tid
    }
    else
    {
        free the memory which has just been allocated for the new timer structure
        an exception has occurred, call the panic function
    }
}
```

A3.3.4 timed_event_at

The *timed_event_at* function is similar to the *timed_event_in* function. The only difference is that instead of asking the Scheduler to call a function after a period has passed, it is asked to call a function at, or after, a particular system time.

```
tid timed_event_at(TIME when, void (*fn) (uint16_t, void *), uint16_t fniarg, void *fnvarg)
{
     calculate difference in timer ticks, between when and the current value of the system clock
     call the function create_timer, which does the actual work.
     return the timer identifier tid obtained from the function create_timer
}
```

Please note that the *when* is stated in microseconds and that the Nucleus PLUS timers use timer ticks.

(The *timed_event_at* function is provided for completeness, and finds little use.)

A3.3.5 cancel_timed_event

The *cancel_timed_event* function asks the Scheduler to attempt to prevent the execution of the timed event, identified with the *eventid*. The *cancel_timed_event* function searching through the linked list of timers, in order to see if the given *eventid* parameter, match any of the timer id contained in the list. If it finds a timer Id that matches, it prevents the execution of this timed event by: deleting the Nucleus timer, freeing the memory allocated for it, and finally by removing the timer structure from the linked list. The *cancel_timed_event* function looks like this:

```
bool_t cancel_timed_event(tid eventid, uint16_t *pmi, void **pmv)
{
        go through the internal timer link list
        {
                if event id equals the timer id in list
                {
                        delete the Nucleus timer
                        remove the timer id from the link list and update the list
                        free the memory allocated for it
                        return TRUE
                }
        }
        return FALSE
}
```

As illustrated the *cancel_timed_event* function return TRUE if it succeed to prevent the execution of the timed event, if not it return FALSE.

A3.3.6 get_time

The *get_time* function must return the current system time in microseconds and looks like this:

```
TIME get_time()
{
retrieve the current value of the Nucleus system timer tick counter
convert the obtained value to microseconds and return it
}
```

A3.4 Function for Message Passing

The Scheduler is responsible for handling message passing between the Scheduler tasks as well as between the Scheduler and other Nucleus tasks, such as the Application task. The functions that need to be considered when porting message passing are:

- put_message
- cancel_message
- get_message
- put_message_in
- put_message_at
- cancel_timed_message

Please refer to [API] for a detailed description of these functions, and for the example functions, please refer to *sched.c* and *sched_private.h*.

A3.4.1 The Schedulers Message Queues

As described, the Scheduler consists of a set of tasks passing messages to each other, where each Scheduler task is attached to one message queue, as illustrated in Figure 7.

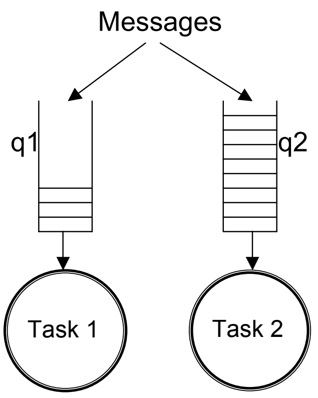


Figure 7: Each Scheduler task is attached to one message queue

The queue messages are held as a FIFO queue, so there is no hard limit to the length of a queue. However, each message consumes memory, thus task implementations prevent messages to build up in the queues. In *sched_private.h* a queue message is simply defined as a structure called *MessageQueueEntryType*, which consists of four elements:

- An unsigned integer *event*, holding the *mi* parameter obtained from the put_message functions
- A void pointer *message*, holding the **mv* parameter obtained from the put_message functions
- A unique message identifier *id*, which can be used to prevent delivery of a message
- A *MessageQueueEntryType* pointer *next*, pointing at the next message in the queue. If it is the last message in the queue, the pointer value is NULL

A3.4.2 put_message

The *put_message* function is used to place a message either to one of the Scheduler message queues, or to the external Nucleus queue, e.g. the queue dedicated to the Application task. If the message is placed on one of the Scheduler message queues, the memory for the new message structure is allocated, the parameters in the structure are set, and the message is placed on the back of the message queue *q*. The *put_message* function returns the message identifier *msgid,* which can be used with the *cancel_message*, refer to section A3.4.3. The *put_message* function will look like this:

```
msgid put_message(qid q, uint16_t mi, void *mv)
{
    obtain an unused message identifier, msgid
    call the function do_put_message, which does the actual work
    return the message identifier msgid
}
```

BlueCore™ Host SW

As illustrated, the *do_put_message* function does the actual work of the *put_message*. It accepts the message identifier to be used in its *id* argument and will look like this:

```
static void do_put_message(qid q, uint16_t mi, void *mv, msgid id)
{
    if  the message queue q is one of the Scheduler message queues
    {
         if q is invalid
        {
             an exception has occurred, call the panic function
        }
        allocate and set the memory for the new message structure. Place it on the back of the
        message queue q. The number of messages on the Scheduler task queues is increased by one
    }
    else the message queue q must refer to the external Nucleus queue dedicated to the application task
    {
         try to place the message at the back of the specified Nucleus queue. This service must
         return immediately regardless of whether or not the request can be satisfied.

         if the request could not be satisfied
         {
              an exception has occurred, call the panic function
         }
    }
}
```

The argument for letting the *do_put_message* function do the actual work, is that the *put_message_in*, see section A3.4.5, and *put_message_at*, see section A3.4.6, functions take advantage of this functionality indirectly.

A3.4.3 cancel_message

The *cancel_message* attempts to prevent delivery of the message with message identifier *mid*, which was previously sent onto the Scheduler message queue *q*. If the message is caught in time the function returns TRUE, otherwise it returns FALSE.

```
bool_t cancel_message(qid q, msgid mid, uint16_t *pmi, void **pmv)
{
    if  the message queue q is one of the Scheduler message queues
    {
        if q is invalid
        {
             an exception has occurred, call the panic function
        }
        try to find the message with message identifier mid, in the message queue q

        if the message in found
        {
             set the pmi and pmv parameters
             the number of messages on the Scheduler task queues is reduced by one
             remove the message from the message queue q, and free the memory allocated for it
             the message identified with mid was caught in time, return TRUE.
        }
        else
        {
            the message was not caught in time, return FALSE
        }
    }
    else
    {
         the message queue q is not one of the Scheduler queues, return FALSE.
    }
}
```

BlueCore^{TM} Host SW

Please note that if the *cancel_message* function obtains a message from its queue, it must free the memory allocated for the message structure, see A3.4.2. Please also note that the canceller must pfree any *pmv* message storage.

A3.4.4 get_message

The get*_message* function is used by a Scheduler task to obtain messages from its own message queue *q*. If the calling task is invalid, or tries to obtain a message from a message queue that is not dedicated to it, an exception has occurred and the *panic* function is called, see *panic.h/c*. If these sanity checks pass, the *get_message* function will try to obtain the first message from its message queue, which is a First-In-First-Out (FIFO) queue. If a message is obtained from the message queue the *pmi* and *pmi* parameters are set, the message is removed from the queue, and the function returns TRUE. Otherwise the function returns FALSE. The get*_message* function will look like this:

```
bool_t get_message(qid q, uint16_t *pmi, void **pmv)
{
    if  the message queue q is one of the Scheduler task queues
    {
        if q is invalid
        {
             an exception has occurred, call the panic function
        }
        if  the calling task does not own q
        {
             an exception has occurred, call the panic function
        }
        try to take a message out of the message queue q
        if  a message has taken from the message queue q
        {
             set the pmi and pmv parameters
             the number of messages on the Scheduler tasks queues is reduced by one
             remove the message from the message queue q, and free the memory allocated for it
             a message has been obtained from the message queue q, return TRUE.
        }
        else
        {
             there were no messages obtained from the message queue q, return FALSE.
        }
    }
    else
    {
         an exception has occurred, call the panic function
    }
}
```

Please note that if the get*_message* function obtains a message from its queue, it must free the memory allocated for the message structure, see section A3.4.2. Furthermore, remember that the Scheduler has no intelligence of how callers use the *pmv* argument. If a coder uses *pmalloc* for the *pmv* part of a message and the message is read without picking it up:

*get_message(A_QID, &foo, (void**) NULL);*

then the pmalloc'ed memory will leak.

A3.4.5 put_message_in

The *put_message_in* function asks the Scheduler to deliver the message after at least period *delay* has passed. The function is actually just a combination of the functions: *timed_event_in* and *put_message* and looks like this:

```
msgid put_message_in(TIME delay, qid q, uint16_t mi, void *mv)
{
    allocate memory for the TimedMessageType structure and initialize it.
    call the function timed_event_in, so that the deliver_timed_message function is called after at
    least period delay has past.
    return a unique message identifier, obtained from the timed_event_in function
}
```

What the *put_message_in* function does is that it allocates memory for a *TimedMessageType* structure, which is defined in *sched_private.h*. The *TimedMessageType* structure consists of four elements:

- An unsigned integer *event*, used for holding the *mi* parameter
- A void pointer *message*, used for holding the **mv* parameter
- A queue identifier *q*, used for holding the *q* parameter
- A timer identifier *id*, used for holding the timer id returned by the *timed_event_in* function

After the new allocated *TimedMessageType* structure is initialized it calls the *timed_event_in* function and asks it to call the function *deliver_timed_message()* after at least period *delay* has passed. As argument to the *deliver_timed_message* function the *fniarg* parameter is set to 0 and the *fnvarg* parameter is set to the allocated new *TimedMessageType* structure. The *timed_event_in* function returns an unique timer identifier which the *put_message_in* function returns as the message identifier *msgid*, which can be used with the *cancel_timed_message*, please refer to section A3.4.7.

When the *deliver_timed_message* function is called, this function just calls the

*do_put_message(qid q, uint16_t mi, void *mv, msgid id)*

function, see section A3.4.2, in order to let it do all the work. The parameter given to the function is obtained from the *TimedMessageType* structure which is past to the *deliver_timed_message* function as the *fnvarg* parameter. Finally the *TimedMessageType* structure is free in order to prevent a memory leak. The *deliver_timed_message* function looks like this:

```
static void deliver_timed_message(uint16_t event, void *message)
{
    cast the void message pointer to a TimedMessageType type
    call the do_put_message
    pfree(timedMessage) to prevent a memory leak
}
```

A3.4.6 put_message_at

The *put_message_at* - is similar to the *put_message_in* function. The only difference is that instead of delivering the message after a period has passed it delivers the message at or after a specific system time *when* has passed, by calling the *timed_event_at* - instead of the *timed_event_in* function. The *put_message_at* function will look like this:

```
msgid put_message_at(TIME when, qid q, uint16_t mi, void *mv)
{
    allocate memory for the TimedMessageType structure and initialize it.
    call the function timed_event_at, so that the deliver_timed_message function is called after a
    period when has past.
    return a unique message identifier obtained from the timed_event_at function
}
```

(The *put_message_at* function is provided for completeness, and finds little use.)

A3.4.7 cancel_timed_message

The *cancel_timed_message* function can be used to attempt to prevent delivery of a message with identifier *mid*, which is previously scheduled to be delivered to message queue *q*. Because the previous timed message request is a combination of the *timed_event_in*/at function call and the *put_message* function call, the message that needs to be cancelled can either be placed in the linked list of timers, or it can be placed on the Scheduler message queue *q*. This demands that the *cancel_timed_message* function first searches for the message in the linked list of timers by calling the *cancel_timed_event* function, and if this function returns FALSE, it then has to search for the message in the Scheduler queue *q*, by calling the *cancel_message* function. If the message is caught in time the function returns TRUE, otherwise it returns FALSE. The *cancel_timed_message* function looks like this:

```
bool_t cancel_timed_message(qid q, msgid mid, uint16_t *pmi, void **pmv)
{
    call cancel_timed_event function with id and function parameter
    if cancel_timed_event returns TRUE the message is found on the linked list of timers
    {
        set the pmi and pmv parameters
        the message identified with mid was caught in time
        free the memory allocated
        return TRUE.
    }
    else call cancel_message with id and function parameter
    {
        if cancel_message function return TRUE the message is found on the Scheduler queue q
        {
        return TRUE
        }
        The message has not caught in time return FALSE
    }
```

A3.5 Memory Handling

The functions that need to be considered when porting the memory management are:

- pmalloc
- zpmalloc
- pfree

Please refer to [API] for a detailed description of these functions, and for the example functions please refer to *pmalloc.c*.

Depending on the environment to which BCHS is ported, memory management can be implemented in two different ways:

1. Use pre-allocated memory with memory allocated from a set of pools. Hereby the pre-allocated memory is used for *pmalloc* and the developer must assure that there is sufficient memory allocated in the pools.
2. Use a dynamic memory allocation scheme, so the *pmalloc* function is analogous to the *malloc* call defined is ANSI C.

In this example the use of dynamic memory allocation and de-allocation are used, because it is estimated that this scheme is sufficient efficiently implemented in the Nucleus PLUS kernel.

A3.5.1 pmalloc

The *pmalloc* function tries to allocate a block of memory from the dynamic memory pool specified in the *Application_Initialize* function. If allocation succeeds the function returns a pointer to the allocated memory, otherwise an exception has occurred and the *panic* function with an error code is called. The *pmalloc* function looks like this:

```
void *pmalloc(uint32_t size)
{
    if size < MIN_ALLOCATE_MEMORY
    {
        size equal MIN_ALLOCATE_MEMORY
    }
    allocate memory in Nucleus with the specified size, return status and a pointer to the allocated
    memory block

    if allocation does not succeed
    {
        call panic with error code
    }
    return pointer to allocated memory block
}
```

A3.5.2 zpmalloc

The *zpmalloc* function is similar to the *pmalloc* function. The only difference is that the allocated memory block is set to zero. The *zpmalloc* function looks like this:

```
void *zpmalloc(uint32_t size)
{
    call the pmalloc function
    set the memory obtained to zero
    return pointer to allocated memory block
}
```

A3.5.3 pfree

The *pfree* function returns the memory, which was previously allocated by either the *pmalloc* - or *zpmalloc* function. The *pfree* function looks like this:

```
void pfree(void *ptr)
{
    if the ptr pointer != NULL deallocate the ptr pointer
}
```

A3.6 The Lower Layer UART Interface

This chapter gives an example of how to port the UART interface for BlueCore™ communication to the Nucleus PLUS kernel. The communication protocol provided by BCHS is a version of the BCSP protocol called ABCSP. Detailed information about ABCSP can be found in [YABCSP].

Please note that the Nucleus PLUS serial driver is used in order to keep the example more generic. It has however, been necessary to make minor modifications in order to adjust it for this purpose. The modifications are explained in section A3.6.4. For more information about the Nucleus PLUS serial driver, please refer to [NU_DRIVER].

A3.6.1 Providing the UART Driver API in the Scheduler

Because a Scheduler has been implemented in this porting example, the porting job for ABCSP is simplified. In this case only the UART driver interface must be provided. The functions that need to be considered when porting the UART interface in this example are:

- UartDrv_Configure
- UartDrv_RegisterHandlers
- UartDrv_TX
- UartDrv_RX

As regards the example function source code, please refer to *uart.c* and *uart.h*.

A3.6.2 UartDrv_Configure

The *UartDrv_Configure* function is called from the main function, see section A3.1 and is used to initialize the communication port. For information of how to initialize the Nucleus PLUS serial driver, please refer to [NU_DRIVER].

A3.6.3 UartDrv_RegisterHandleres

Seeing that the Scheduler is not re-entrant safe, foreground tasks, such as the UART communication drivers, must not interrupt the Scheduler background job. Instead a mechanism must be implemented to inform the background task that a foreground task needs attention. The functions that need consideration as to background interrupt handling are:

- register_bg_int
- bg_intx, where x is the background interrupt number

Please note that only the background interrupt number 1, 2 and 6 are considered in this porting example, because these are used by the UART communication driver for ABCSP. As regards the example function source code please refer to *bg_int.c* and *bg_int.h.*

The *UartDrv_RegisterHandlers* function is called from the main function, see section A3.1, and is used to register the background interrupts by calling *register_bg_int,* which defines the background interrupt number and the function that needs to be called when the Scheduler receives this number. The *UartDrv_RegisterHandlers* function will look like this:

```
void UartDrv_RegisterHandlers(void)
{
        register_bg_int(1, UartDrv_Rx);
        register_bg_int(2, abcsp_pumptxmsgsOut);
#ifdef ROM_BUILD_ENABLE
        register_bg_int(6, abcsp_restart);
#endif /*ROM_BUILD_ENABLE*/
}
```

where background interrupt number:

1 is used to inform the Scheduler that data is available in the Nucleus RX buffer, and that it needs to call the function *UartDrv_Rx*, see section A3.6.4.

2 is used to inform the Scheduler that there are data needing to be passed to the Nucleus UART driver, which demands that it calls the function *abcsp_pumptxmsgsOut*, For a more thorough description of this issue please refer to section A3.6.5. Please note that *abcsp_pumptxmsgsOut* is a part of the ABCSP protocol source code, and nothing needs to change in this function.

6 is used to inform the Scheduler that the system has restarted doing initialisation of the system, and that it needs to call the function *abcsp_restart*. Please note that this background interrupt is only used if the bootstrap function for the ROM version of the chip is used. For more information about the bootstrap function, please refer to [BOOT ROM].

A3.6.4 UartDrv_RX

When the UART driver receives a message from the Bluetooth Core stack it must inform the Scheduler. As described previously the UART driver is not allowed to interrupt the Scheduler background job, instead it uses background interrupt number 1 for this purpose.

In this example the UART driver provided by Nucleus PLUS is used in order to keep the example more generic. However, it has been necessary to make minor modifications to the Nucleus PLUS SDC_LISR routine. The SDC_LISR routine is part of the Nucleus PLUS source code and can be found in *sdc.c.* One of the modifications that have been made is that every time the SDC_LISR routine receives the character 0xC0 twice on the com port define in the *UartDrv_Configure* function it activates a high level interrupt (HISR). This HISR being activated is named bgRxIRQ, and is defined in the *Application_Initialize* function, see chapter A2. In order for the Scheduler to be re-entrant safe, the bgRxIRQ only calls a function called *bg_int1*, which must be implemented by the developer.

In this porting example the *bg_int1* function makes a background interrupt by making a Nucleus PLUS message, and place it in front of the Nucleus queue dedicated to the Nucleus Scheduler task. The *bg_int1* function can be found in *bg_int.c* and looks like this:

```
void bg_int1 (void)
{
    make a Nucleus PLUS message, which tells the Scheduler that this message must be viewed as
    a background interrupt with the interrupt number 1.
}
```

When the *sched* function obtains this background interrupt message, see section A3.2, it calls the function *UartDrv_RX*, see section A3.6.3, because it now knows that some data are available in the UART driver's RX buffer. The *UartDrv_RX* source code can be found in *uart.c* and looks like this:

```
void UartDrv_Rx(void)
{
    while there are some data on the Nucleus UART RX buffer, copy it into the local RX buffer.
    call the function abcsp_uart_deliverbytes
    in case data are still in the local RX buffer, call the bg_int1 function
}
```

Please note that it can be preferable to optimise the Nucleus PLUS UART driver, so that the data is taken directly from the Nucleus RX buffer instead of copying it into a local RX buffer.

A3.6.5 UartDrv_TX

When the host (BCHS) sends a message to the Bluetooth Core stack, ABCSP repeatedly calls the macro ABCSP_REQ_PUMPTXMSGS to translate the message into its BCSP wire format and push these bytes out using the *UartDrv_TX* function.

The ABCSP_REQ_PUMPTXMSGS macro calls a function named *abcsp_req_pumptxmsgs*, see *config_event.h*. When porting the lower layer UART interface it is possible to redefine this macro to another function name and implement it, thus it provides the same functionality as the *abcsp_req_pumptxmsgs* function. However, the most simple way to do this is to just use the *abcsp_req_pumptxmsgs* function, which can be found in *abcspRdHl.c.* The reason for this is that then the developer only needs to implement a function called *bg_int2*, instead of rewriting the *abcsp_req_pumptxmsgs* function in the ABCSP source code. The *abcsp_req_pumptxmsgs* must not call back into functions in the YABCSP software.

The purpose of the *bg_int2* function is to inform the Scheduler that data need to be sent to the UART driver. As explained in section A3.6.3, the Scheduler uses background interrupt 2 for this. The *bg_int2* function source code can be found in *bg_int.c* and it looks like this:

```
void bg_int2 (void)
{
    make a Nucleus PLUS message, which tells the Scheduler that this message must be viewed as
    a background interrupt with the interrupt number 2.
}
```

As illustrated the *bg_int2* function makes a background interrupt 2 by placing a message in front of the Nucleus queue, which is dedicated to the Nucleus Scheduler task. When the *sched* function obtains this background interrupt message, see section A3.2, the function *abcsp_pumptxmsgsOut* is called, see section A3.6.3. In this way the Scheduler is informed that data need to be sent to the UART driver without interrupting the Scheduler background job.

Together with other functions the *abcsp_pumptxmsgsOut* function calls the ABCSP_UART_SENDBYTES macro, which is defined in *config_txmsg.h*. This macro calls the *abcsp_uart_sendbytes* function in abcspTxHandler.c, which is important because this function needs to be ported. The easiest way to do this is to use the *abcsp_uart_sendbytes* function, because the developer then only needs to implement a function called *UartDrv_TX*, instead of changing the function in the ABCSP source code.

In this case *UartDrv_TX* must pass *num_to_send* bytes from *buf* onto the UART TX buffer, and *numSend* holds the number of bytes put onto the TX queue after the function returns.

*bool_t UartDrv_Tx(char *buf, uint16_t num_to_send, uint16_t *numSend)*

If the bytes are passed to the UART TX buffer the function must return true else false.

In this porting example the source code of the *UartDrv_Tx* function can be found in *uart.c*, and it looks like this:

```
bool_t UartDrv_Tx(char *buf, uint16_t num_to_send, uint16_t *numSend)
{
    pass num_to_send bytes from buf onto Nucleus TX buffer by using the Nucleus PLUS function
    call NU_SD_Put_Char
    return TRUE if any of the bytes pass, else FALSE.
}
```

A4. Document References

Document:	Document number:	Reference:
BlueCore™ Host Software User Guide	BCHS-GU-001_UserGuide	[USERG]
YABCSP – Yet Another BCSP stack	Technical Communications Style Guidelines FEB02	[YABCSP]
Scheduler API	BCHS-API-009_ScedulerApi	[API]
BlueCore Serial Protocol (BCSP)	AN003, AN004, AN005	[BCSP]
License Agreement	BCHS-ME-003_LicenceAgreementRfcomm BCHS-ME-002_LicenceAgreementEval BCHS-ME-007_LicenceAgreementHci	[LIC]
Bluetooth® Core Specification		[BT11]
Nucleus PLUS reference manual	Rev. 104 May 20 2002	[NU]
Nucleus PLUS Serial Driver reference manual	Rev. 101	[NU_DRIVER]
BC COMMAND API	BCHS-API-017_BcCmdApi	[BOOT_ROM]

All documents are not part of the BCHS delivery, but can be found on the CSR website: www.csr.com.

BlueCore™ Host SW