SEARAN

2017-06-18

dotstack™

dotstack integration with STM32F4 & FreeRTOS.

Contents

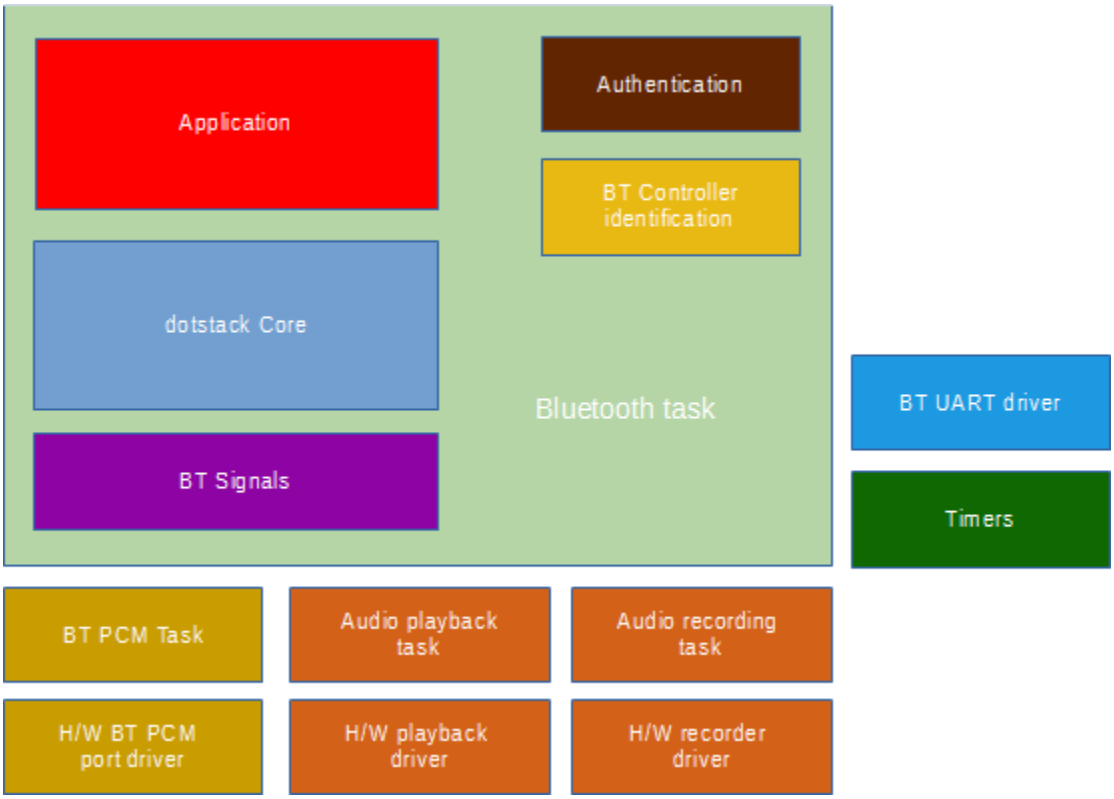

Figure 1

The general structure of the integration layer is shown in the Figure 1. The layer consists of the following components:

- Bluetooth task
- Application
- dotstack Core library (dotstack Core)
- Implementation of dotstack signals processing interface (BT Signals)
- Bluetooth controller PCM task (BT PCM task)
- Bluetooth controller hardware PCM interface driver (H/W BT PCM port driver)
- Audio playback task
- Audio playback hardware interface driver (H/W playback driver)
- Audio recording task
- Audio recording hardware interface driver (H/W recorder driver)
- Bluetooth controller HCI interface driver (BT UART driver)
- Implementation of dotstack timers interface (Timers)
- Authentication
- BT Controller identification

1. Bluetooth Task

This task is the central piece of the integration layer which runs the stack and also glues all other components together. The task is defined in common/btstask.h and common/btstask.c. On startup the task initializes BT UART driver and then goes to an infinite loop where it listens for a semaphore. There is a mask of flags where each bit corresponds to a “signal”. Each signal has a function associated with it called signal handler. Once the task gets ownership of the semaphore it checks the mask and calls handlers of all signals that are currently set. The mask is then reset. This process repeats indefinitely.

2. Bluetooth controller UART driver

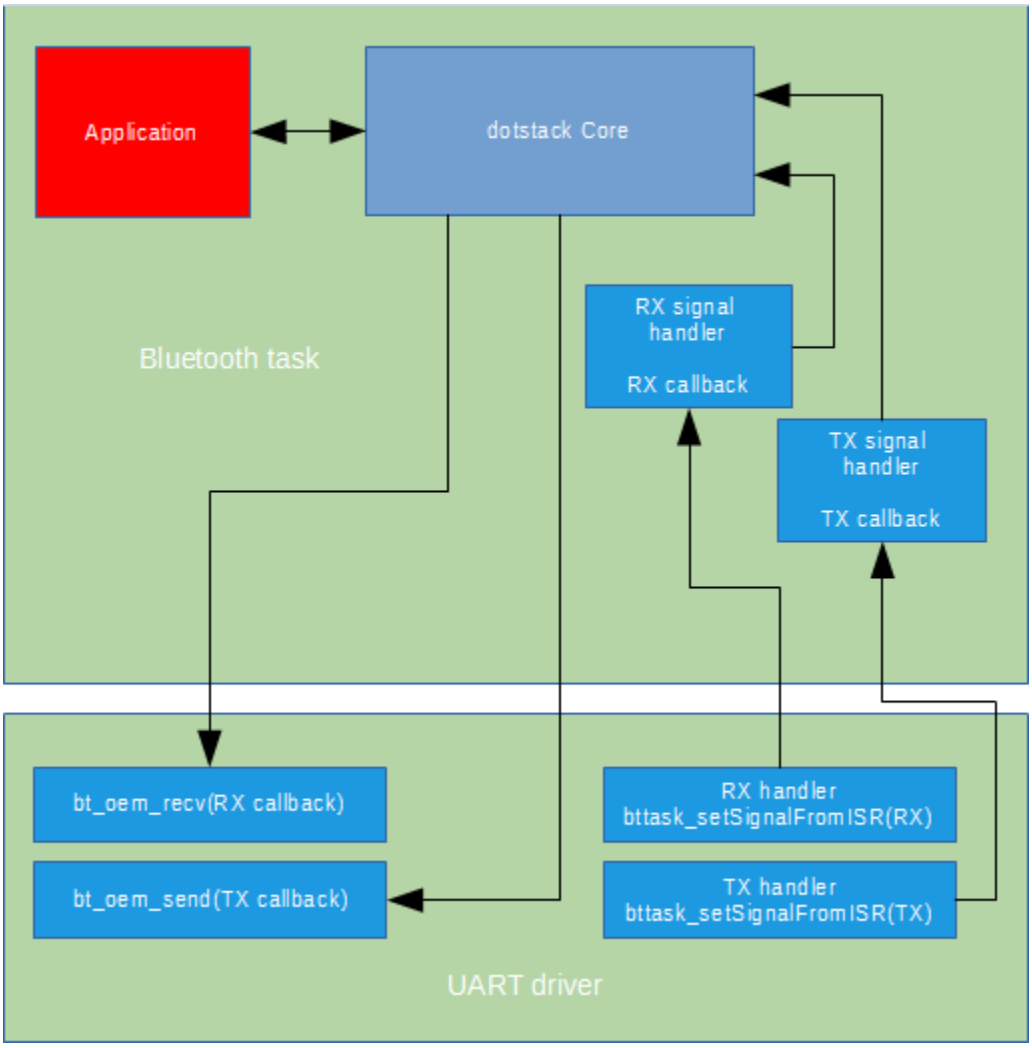

Figure 2

This driver implements communication interface between the stack and the BT controller. Although the stack can work with any controller that supports HCI interface, controllers from different vendors often require some specific configuration before they can be used. The driver also performs this task.

For each supported controller there is at least one variant of the driver. Drivers are implemented in files that begin with “btport”. For example, for CSR881x controllers there are 2 variants of the driver – common/btport_csr881x.c (uses UART interrupts) and common/btport_csr881x_dma.c (uses DMA). Which variant of the driver to use is usually chosen by setting an appropriate option in common/config.h. For example, if “BT_UART_USE_DMA” is defined in common/config.h then the DMA version will be used, otherwise – the interrupt version will be used.

This communication interface between the stack and the HCI controller consists of two functions:

- void bt_oem_send(const bt_byte* buffer, bt_uint len, bt_oem_send_callback_fp callback);
- void bt_oem_recv(bt_byte* buffer, bt_uint len, bt_oem_recv_callback_fp callback);

When the stack wants to send something it calls bt_oem_send() passing a pointer to the buffer to be sent, the length of the buffer and a pointer to a callback function. The driver stores these values and starts asynchronous transmission using either UART interrupt or DMA. On the specified number of bytes has been sent the driver sets the BTTASK_SIG_TX signal. This tells the BT task that on the next iteration it has to call the BTTASK_SIG_TX signal handler - bttask_pal_handleTxSignal() – which is defined in the drivers source file (e.g., common/btport_csr811_dma.c). The handler calls the callback function passed when bt_oem_send() was called. This technique ensures that the callback is called in the context of the BT task.

When the stack needs data from the controller is calls bt_oem_rcv() passing a pointer to the buffer where received data should be stored, the number of bytes to receive and a pointer to a callback function. The driver stores these value and start asynchronous receive using either UART interrupts or DMA. Once the specified number of bytes has been received the driver sets the BTTASK_SIG_RX signal. This tells the BT task that on the next iteration it has to call the BTTASK_SIG_RX signal handler - bttask_pal_handleRxSignal() – which is defined in the drivers source file (e.g., common/btport_csr811_dma.c). The handler calls the callback function passed when bt_oem_recv() was called. The callback should be called only when the exact number of bytes specified in the bt_oem_recv() call has been received.

3. Audio playback and recording

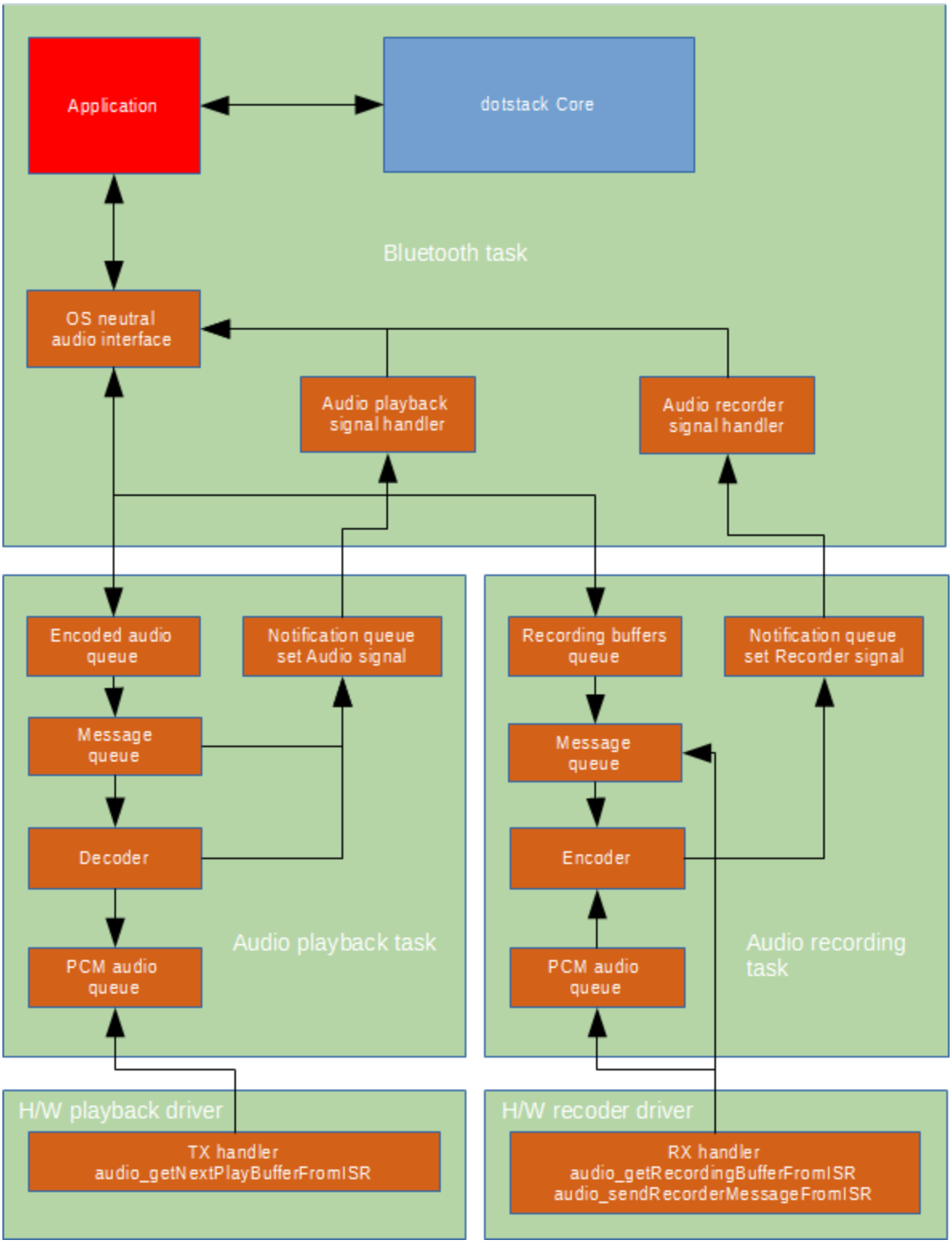

Figure 3

3.1. Audio playback

The audio playback interface is comprised of 3 layers of abstraction:

- OS-independent audio playback interface
- OS-specific audio playback interface (Audio playback task in Figure 3)
- Hardware-specific audio playback interface (H/W playback driver in Figure 3)

The OS-independent interface is defined in common/btaudio.h. It consists of a set of functions that are named “btaudio_<operation name>”:

void btaudio_init(btaudio_Callback callback, btaudio_Callback recorderCallback);

bt_bool btaudio_configurePlayback(btaudio_CodecConfig* config);

bt_bool btaudio_configureAnalizer(btaudio_CodecConfig* config);

void btaudio_startPlayback(void);

void btaudio_stopPlayback(void);

void btaudio_pausePlayback(void);

void btaudio_resumePlayback(void);

void btaudio_playPacket(btaudio_MediaPacket* packet);

void btaudio_analizeStream(btaudio_MediaPacket* packet);

void btaudio_setVolume(bt_uint volume, bt_bool fromISR);

void btaudio_setVolumeEx(bt_uint volume, bt_bool fromISR);

These functions is what Application has to use when it needs to play audio received from the stack or control the playback. The interface is implemented in common/btaudio.c. The implementation is suitable for any OS or no-OS integration.

The OS-specific interface is also defined in common/btaudio.h. It consists of a set of functions that are named “btaudio_pal_<operation name>”:

void btaudio_pal_init(btaudio_Callback callback);

void btaudio_pal_configurePlayback(bt_ulong samplingRate, bt_byte codecType, void* codecConfig, bt_byte btAudioInterface, bt_byte audioProcessor);

void btaudio_pal_configureAnalizer(bt_ulong samplingRate, bt_byte codecType, void* codecConfig);

void btaudio_pal_startPlayback(void);

void btaudio_pal_stopPlayback(void);

void btaudio_pal_pausePlayback(void);

void btaudio_pal_resumePlayback(void);

void btaudio_pal_playPacket(btaudio_MediaPacket* packet);

void btaudio_pal_analizeStream(btaudio_MediaPacket* packet);

void btaudio_pal_setVolume(bt_uint volume, bt_bool fromISR);

Implementation of these functions is different under each OS. The FreeRTOS implementation is located in common/audio.c. The OS-specific audio playback interface under FreeRTOS is implemented in a separate task. Communications between the audio task and the BT task happen with the help of 2 queues – message queue and notification queue. The first queue is used to receive commands from the BT task. I.e., when "btaudio_pal_" function is called by the OS-independent interface, usually no actual work is done except putting a message to the message queue. The notification queue is used to notify the BT task that an operation inside the audio task has completed.

The hardware-specific audio playback interface is defined in common/audio.h. It is used to communicate with the real hardware. This interface is actually can be anything because it depends on the available hardware and is used only by the OS-specific interface. In case of FreeRTOS the hardware-specific interface consists of the following functions:

void audio_init(void);

audio_PlayBuffer* audio_getNextPlayBuffer(void);

audio_PlayBuffer* audio_getNextPlayBufferFromISR(portBASE_TYPE *higherPriorityTaskWoken);

void audio_pal_init(void);

void audio_pal_startPlayback(uint32_t samplingRate, uint8_t audioProcessor);

void audio_pal_stopPlayback(void);

void audio_pal_pausePlayback(void);

void audio_pal_resumePlayback(void);

void audio_pal_setVolume(uint16_t volume);

uint32_t audio_pal_getFlag(uint32_t flag);

uint32_t audio_pal_getFlags(void);

Let's see how playing an A2DP media packet works. When an A2DP packet arrives the stack calls the callback registered with bt_a2dp_register_callback() with the event code A2DP_EVT_MEDIA_PACKET_RECEIVED. The parameter to this event is a pointer to a structure that contains encoded audio. A2DP_EVT_MEDIA_PACKET_RECEIVED event handler calls

btaudio_playPacket() which in turn calls btaudio_pal_playPacket(). In this function the packet is added to a list and a message MSG_PLAY_PACKET is sent to the audio playback task's message queue. The task's function reads messages from the queue and when it finds MSG_PLAY_PACKET message it calls fillPlayBuffer(). This function checks if there is already a packet that has not been completely decoded. If there is one it continues decoding this packet. Otherwise it takes next packet from the list.

Decoded PCM frames are stored in another list. When the length of this list (stored in mPlayQueueLen) reaches PLAYBACK_START_THRESHOLD, audio_pal_startPlayback() is called which configures audio CODEC and sends first portion of PCM frames to it using DMA transfer. Once the transfer is complete, the DMA interrupt handler calls audio_getNextPlayBufferFromISR() which returns next portion of PCM frames. While PCM frames are being sent to the CODEC, the audio playback task continues decoding arriving packets. As long as the remote device continues streaming there will always be something in the list of decoded frames.

After the packet with encoded audio has been completely decoded it has to be returned to the application for reuse. This is done through the notification queue. When fillPlayBuffer() is done with the current packet it puts a NOTIF_PACKET_PLAYED message to the notification queue and sets BTTASK_SIG_AUDIO signal. The BT task when sees that signal is set calls its handler – audioSignalHandler(). In this function the notification queue is read and an application callback (set with btaudio_init() call) is executed.

3.2. Audio recording

The audio recording interface is similar to the audio playback one. It is also comprised of 3 layers of abstraction:

- OS-independent audio recording interface
- OS-specific audio recording interface (Audio recording task in Figure 3)
- Hardware-specific audio recording interface (H/W recorder driver in Figure 3)

The OS-independent interface is defined in common/btaudio.h. It consists of a set of functions that are named "btaudio_<operation name>":

```
bt_bool btaudio_configureRecorder(btaudio_CodecConfig* config);

void btaudio_startRecorder(void);

void btaudio_stopRecorder(void);

void btaudio_pauseRecorder(void);

void btaudio_recordPacket(btaudio_MediaPacket* packet);
```

These functions is what Application has to use when it needs to record audio form some source and send it to a remote device or control the recording. The interface is implemented in common/btaudio.c. The implementation is suitable for any OS or no-OS integration.

The OS-specific interface is also defined in common/btaudio.h. It consists of a set of functions that are named "btaudio_pal_<operation name>":

void btaudio_pal_configureRecorder(bt_ulong samplingRate, bt_byte codecType, void* codecConfig, bt_byte btAudioInterface, bt_byte audioProcessor);

void btaudio_pal_initRecorder(btaudio_Callback callback);

void btaudio_pal_startRecorder(void);

void btaudio_pal_stopRecorder(void);

void btaudio_pal_pauseRecorder(void);

void btaudio_pal_recordPacket(btaudio_MediaPacket* packet);

Implementation of these functions is different under each OS. The FreeRTOS implementation is located in common/voice.c. The OS-specific audio recording interface under FreeRTOS is implemented in a separate task. Communications between the recording task and the BT task happen with the help of 2 queues – message queue and notification queue. The first queue is used to receive commands from the BT task. I.e., when "btaudio_pal_" function is called by the OS-independent interface, usually no actual work is done except putting a message to the message queue. The notification queue is used to notify the BT task that an operation inside the recording task has completed.

The hardware-specific audio recording interface is defined in common/voice.h. It is used to communicate with the real hardware. This interface is actually can be anything because it depends on the available hardware and is used only by the OS-specific interface. In case of FreeRTOS the hardware-specific interface consists of the following functions:

audio_RecordBuffer* audio_getRecordingBufferFromISR(portBASE_TYPE *higherPriorityTaskWoken);

void audio_sendRecorderMessageFromISR(RecorderTaskMessage* msg, signed portBASE_TYPE* higherPriorityTaskWoken);

void audio_pal_initRecorder(void);

void audio_pal_configureRecorder(uint32_t samplingRate);

void audio_pal_startRecorder();

void audio_pal_pauseRecorder();

void audio_pal_stopRecorder(void);

Audio recording is used in examples that demonstrate HFP and HSP profiles. It is also used in our custom solution for voice over BLE.

If an app wants to receive audio from an audio source (e.g., microphone) it should constantly supply the recoding layer with packets where the audio will be stored. The app does it by calling btaudio_recordPacket(). This function calls btaudio_pal_recordPacket () which sends a MSG_RECORD_PACKET message to the recording task's queue. A pointer to the packet is passed along

with the message. The task's function reads messages from the queue and when it sees MSG_RECORD_PACKET it adds the packet to a list and sends a MSG_FILL_RECORD_PACKET message. The MSG_FILL_RECORD_PACKET handler calls fillRecordPacket(). This one checks if there is anything in the list of raw PCM frames read from the audio source. As long as this list is not empty, raw PCM frames are encoded and saved to the packet supplied by the app.

When the current recording packet is full it has to be returned to the app. This is done through the notification queue. fillRecordBuffer() puts a NOTIF_PACKET_RECORDED message to the notification queue and sets BTTASK_SIG_RECORDER signal. The BT task, when sees that signal is set, calls its handler – recorderSignalHandler(). In this function the notification queue is read and an application callback (set with btaudio_init() call) is executed. The NOTIF_PACKET_RECORDED causes the application callback to be executed with BTAUDIO_EVENT_PACKET_RECORDED event. The handler of this event usually sends the recorded audio to the remote device by calling an API from dotstack library or, if the BT controller has PCM port, by passing the packet directly to the PCM port interface.

Actual audio recording occurs in H/W recorder driver. There is an array of buffers that the H/W driver uses to store recorded frames. The driver accesses this array with audio_getRecordingBufferFromISR(). This function returns a pointer to the next free buffer and decreases the number of free buffers.

For STM32F4-Discovery board the H/W recorder driver is implemented in common/STM32F4-DISCOVERY/voice_pal.c. When recording is enabled (the app does that by calling btaudio_startRecorder() which eventually leads to an audio_pal_startRecorder() call), each audio frame generates an SPI interrupt. The interrupt handler, if it has yet no pointer to a buffer for storing PCM frames, calls audio_getRecordingBufferFromISR() and stores returned pointer in mBuffer. Once the buffer is full the handler sends a MSG_BUFFER_RECORDED to the recording task. The task increases the number of filled buffers and calls fullRecordBuffer().

4. Bluetooth controller PCM port interface

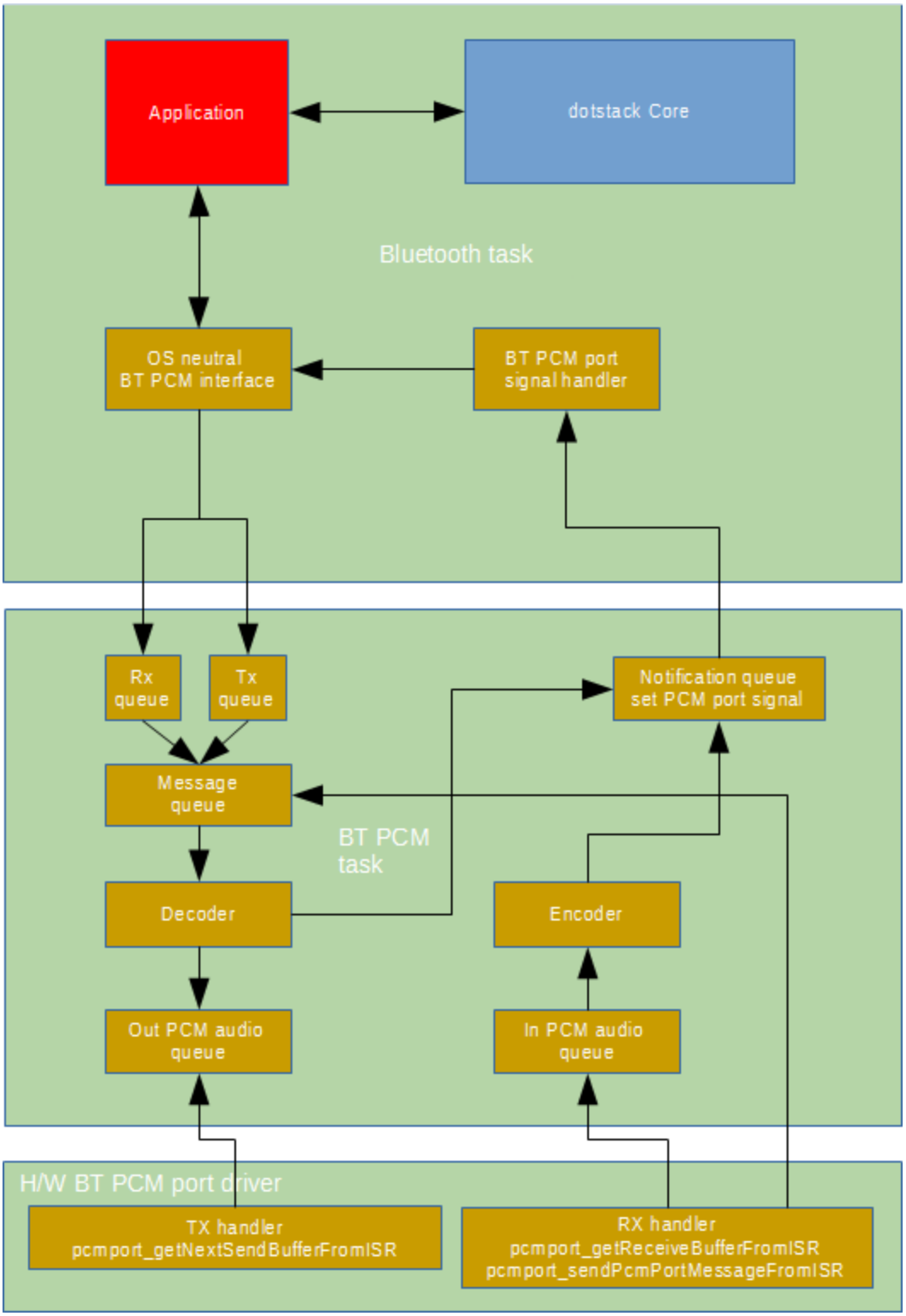

Figure 4

Most BT controllers have a PCM port which is used to transfer audio without involving the HCI interface. With some controllers (e.g., CSR8811) using the PCM port is the only way to implement SCO audio links.

The PCM port interface is similar to audio playback and recording interfaces and is comprised of 3 layers of abstraction:

- OS-independent PCM port interface
- OS-specific PCM port interface (BT PCM task in Figure 4)
- Hardware-specific PCM port interface (H/W BT PCM port driver in Figure 4)

The OS-independent interface is defined in common/btpcmport.h. It consists of a set of functions that are named “btpcmport_<operation name>”:

void btpcmport_init(btpcmport_Callback callback);

bt_bool btpcmport_configure(btaudio_CodecConfig* config);

void btpcmport_start(void);

void btpcmport_stop(void);

void btpcmport_receivePacket(btaudio_MediaPacket* packet);

void btpcmport_sendPacket(btaudio_MediaPacket* packet);

void btpcmport_connect(bt_uint hconn, bt_id cid);

void btpcmport_disconnect(void);

These functions is what Application has to use when it needs to send to or receive from the controller’s PCM port. The interface is implemented in common/btpcmport.c. The implementation is suitable for any OS or no-OS integration.

The OS-specific interface is also defined in common/btpcmport.h. It consists of a set of functions that are named “btpcmport_pal_<operation name>”:

void btpcmport_pal_init(btpcmport_Callback callback);

void btpcmport_pal_configure(bt_ulong samplingRate, bt_byte codecType, void* codecConfig, bt_byte btAudioInterface, bt_byte audioProcessor);

void btpcmport_pal_start(void);

void btpcmport_pal_stop(void);

void btpcmport_pal_receivePacket(btaudio_MediaPacket* packet);

void btpcmport_pal_sendPacket(btaudio_MediaPacket* packet);

void btpcmport_pal_connect(bt_uint hconn, bt_id cid);

void btpcmport_pal_disconnect(void);

Implementation of these functions is different under each OS. The FreeRTOS implementation is located in common/pcmport.c. The OS-specific PCM port interface under FreeRTOS is implemented in a separate task. Communications between the PCM port task and the BT task happen with the help of 2 queues – message queue and notification queue. The first queue is used to receive commands from the BT task. I.e., when "btpcmport_pal_" function is called by the OS-independent interface, usually no actual work is done except putting a message to the message queue. The notification queue is used to notify the BT task that an operation inside the PCM port task has completed.

The hardware-specific PCM port interface is defined in common/pcmport.h. It is used to communicate with the real hardware. This interface is actually can be anything because it depends on the available hardware and is used only by the OS-specific interface. In case of FreeRTOS the hardware-specific interface consists of the following functions:

void pcmport_pal_init(void);

void pcmport_pal_configure(bt_ulong samplingRate, bt_byte codecType, void* codecConfig);

void pcmport_pal_connect(bt_uint hconn, bt_id cid);

void pcmport_pal_disconnect(void);

void pcmport_pal_startSending(void);

void pcmport_pal_stopSending(void);

void pcmport_pal_startReceiving(void);

void pcmport_pal_stopReceiving(void);

The implementation of this interface for CSR8811 is located in common/pcmport_pal_csr881x.c.

4.1. Sending audio data to PCM port

When the app wants to send an audio packet (e.g., when it receives one from the audio recorder) to a remote device through the PCM port it calls btpcmport_sendPacket() which in turn calls btpcmport_pal_sendPacket(). In this function the packet is added to a list and a message MSG_SEND_PACKET is sent to the PCM port task's message queue. The task's function reads messages from the queue and when it sees MSG_SEND_PACKET message it calls fillSendBuffer(). This function checks if there is already a packet that has not been completely sent. If there is one it continues sending this packet. Otherwise it takes next packet from the list.

Before the audio data can be sent to the controller it has to be decoded and stored into another buffer. Under FreeRTOS "decoding" is simply copying the data from the media packet to the send buffers because media packets provided by the app are already in correct format.

Filled send buffers are stored in another list. When the length of this list (stored in mSendQueueLen) reaches NUM_SEND_BUFFERS, a NOTIF_START_SENDING message is put to the notification queue and BTTASK_SIG_SCOPORT signal I set. The BT task when sees that signal is set calls its handler – pcmPortSignalHandler (). The signal handler reads notification queue and when it sees

NOTIF_START_SENDING message it calls pcmport_pal_startSending(). This function gets a buffer from the list of filled send buffers and sends it to the controller using DMA transfer. Once the transfer is complete, the DMA interrupt handler calls pcmport_getNextSendBufferFromISR () which returns next send buffer. While audio frames are being sent to the controller, PCM port task continues "decoding" arriving packets. As long as the app continues supplying media packets there will always be something in the list of send buffers.

After the media packet has been completely sent it has to be returned to the application for reuse. This is done through the notification queue. When fillSendBuffer() is done with the current packet it puts a NOTIF_PACKET_SENT message to the notification queue and sets BTTASK_SIG_SCOPORT signal. The BT task, when sees that signal is set, calls its handler – pcmPortSignalHandler (). In this function the notification queue is read and an application callback (set with btpcmport_init() call) is executed.

4.2. Receiving audio data from PCM port

If the app wants to receive audio from the PCM port it should constantly supply the PCM port interface with packets where the audio will be stored. The app does it by calling btpcmport_ receivePacket (). This function calls btpcmport_pal_ receivePacket () which sends a MSG_RECEIVE_PACKET message to the PCM port task's queue. A pointer to the packet is passed along with the message. The task's function reads messages from the queue and when it sees MSG_RECEIVE_PACKET it adds the packet to a list and calls fillReceivePacket(). This function checks if there is anything in the list of received audio frames. As long as this list is not empty, audio frames are encoded and saved to the packet supplied by the app.

"Encoding" is simply copying the data from the receive buffer to the media packet because data from the controller is already in the correct format.

When the current receive packet is full it has to be returned to the app. This is done through the notification queue. fillReceiveBuffer() puts a NOTIF_PACKET_RECEIVED message to the notification queue and sets BTTASK_SIG_SCOPORT signal. The BT task, when sees that signal is set, calls its handler – pcmPortSignalHandler (). In this function the notification queue is read and an application callback (set with btpcmport_init() call) is executed. The pcmPortSignalHandler causes the application callback to be executed with BTSCOPORT_EVENT_PACKET_RECEIVED event. The handler of this event sends the received audio to the audio playback interface.

Audio from the controller is received in H/W BT PCM port driver. There is an array of buffers that the H/W driver uses to store received frames. The driver accesses this array with pcmport_getReceiveBufferFromISR (). This function returns a pointer to the next free buffer and decreases the number of free buffers. Once the buffer is filled with audio frames a MSG_BUFFER_RECEIVED message is sent to the PCM port task. The MSG_BUFFER_RECEIVED handler increases the number of filled buffers and calls fillReceivePacket ().

5. Signal processing

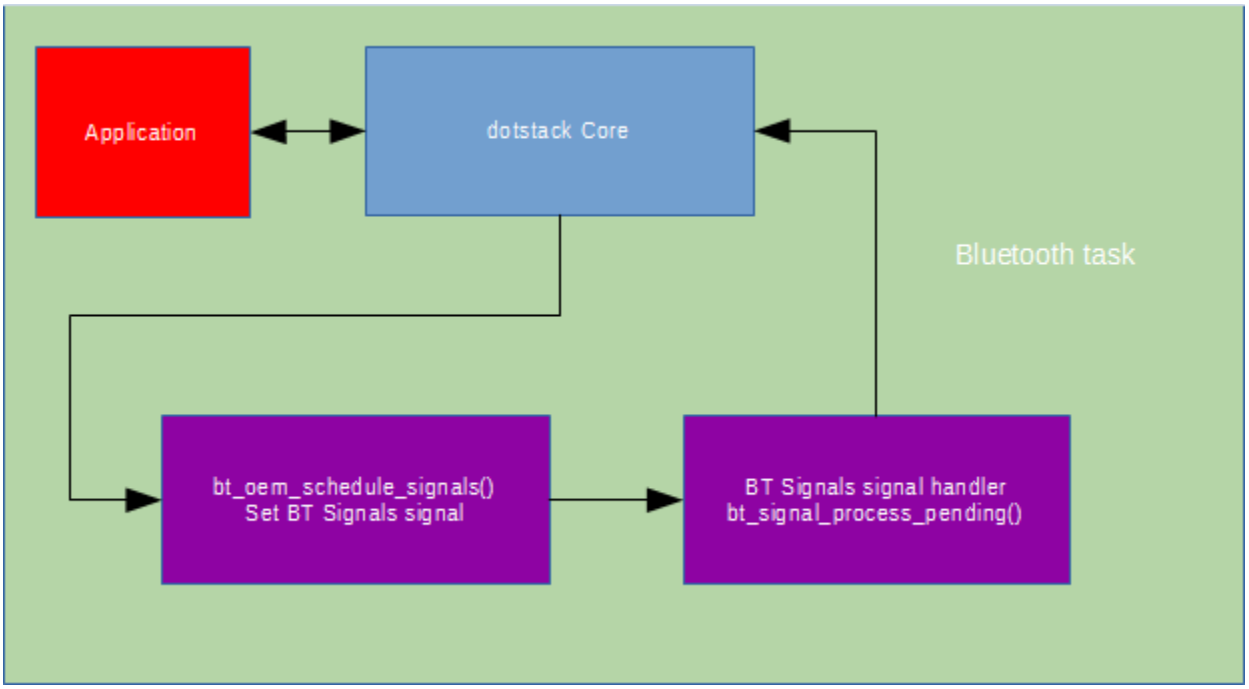

Figure 5

Almost everything in the stack is done asynchronously. To communicate between various parts of the stack we use signals. E.g., when bt_spp_send (sends a buffer over SPP) is called, it does not send anything to the controller. It simply saves the pointer and the length of the buffer to be sent and sets a signal. The signal has a function associated with it. This function has to be called at some point in time and this should happen as soon as possible. But this cannot be done right after the signal has been set because processing a signal may set another signal (it may be the same signal). This may create a chain of calls that will eventually eat up all program stack or (if the stack is large) will not allow other parts of the dotstack to run. To solve this problem we defined the signal processing interface that has 2 function:

- void bt_oem_schedule_signals(void)
- void bt_signal_process_pending(void)

When the stack calls bt_oem_schedule_signals() this function simply sets the BTTASK_SIG_BTSIGNAL signal. On the next iteration of the task's loop bt_signal_process_pending() is called which go through the list of signals and calls handlers of all set signals.

6. Timers

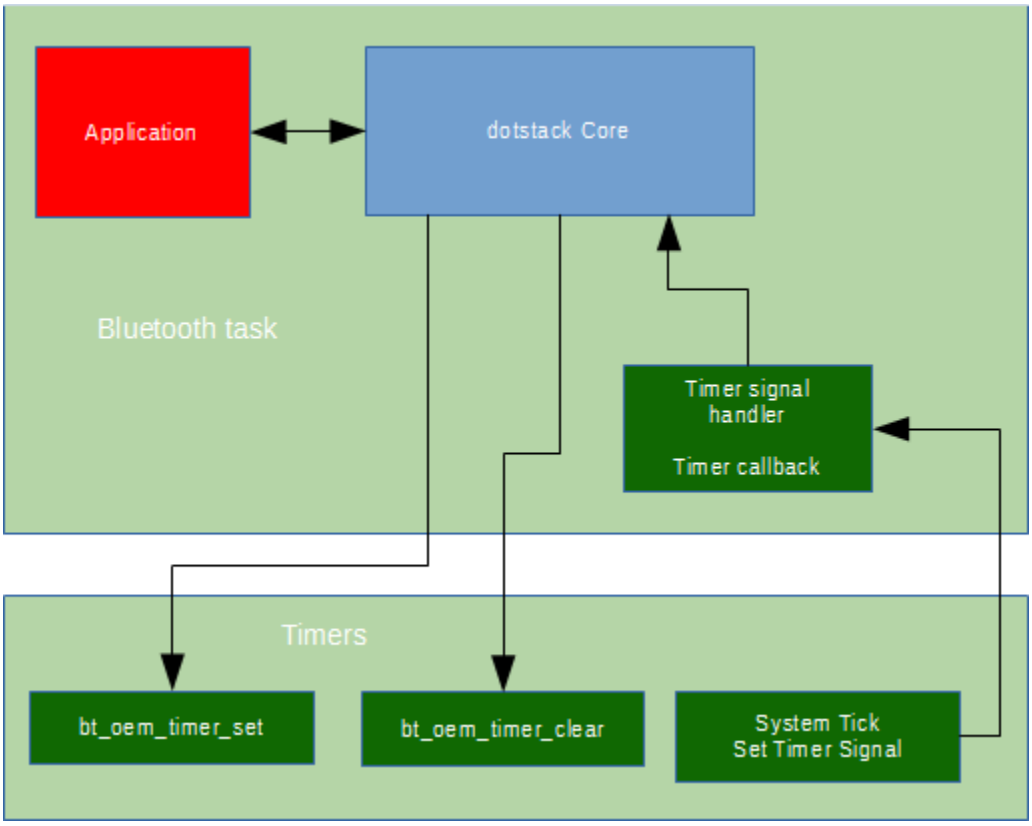

Figure 6

Timers are used to implement various time outs. This interface has 2 functions:

- void bt_oem_timer_set(bt_uint timerId, bt_ulong milliseconds, bt_timer_callback_fp callback);
- void bt_oem_timer_clear(bt_uint timerId);

Under FreeRTOS timers are implemented in common/bttimer.c.

When the stack calls bt_oem_timer_set() this function converts “milliseconds” to system ticks and stores this value and timer’s callback into an array. Then it finds a timer with the earliest expiration time and updates mTimerCounter with the number of ticks left until the earliest timer is to expire.

mTimerCounter is decreased on every inforcation bttimer_onSystemTick() which is called from the system tick hook. When mTimerCounter reaches 0 the BTTASK_SIG_TIMER signal set. The handler of this signal (bttask_pal_handleTimerSignal) goes through the list of timers and calls callback function of all expired timers.

When the stack calls bt_oem_timer_clear(), the specified timer is stopped by setting its duration to 0. Then mTimerCounter is updated so it contain the number of ticks until expiration of the earliest timer.

Timer used by the stack have very low requirements. The resolution of 100 ms is just enough. Also timers do not have to be very accurate.

7. Authentication

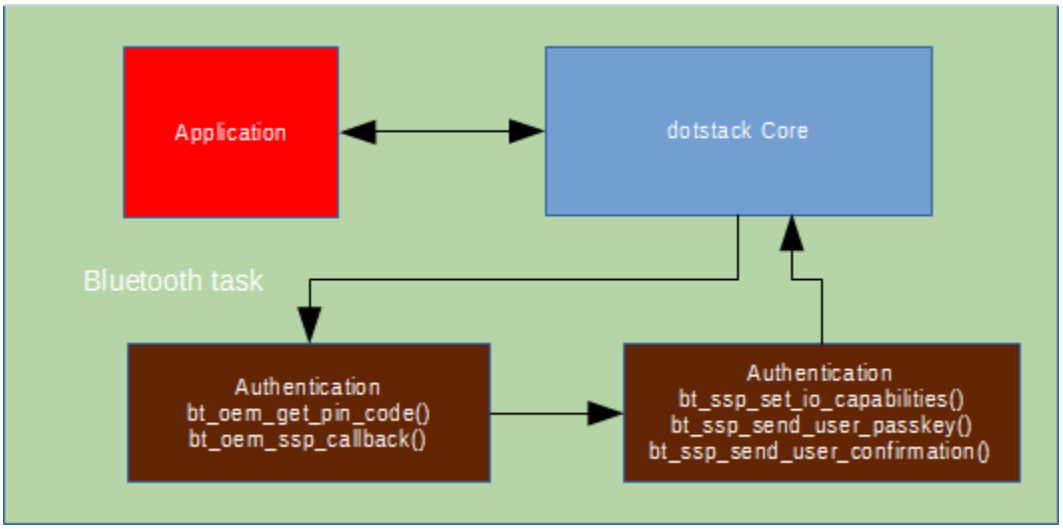

Figure 7

This interface is used when two controllers initiated a pairing procedure. There are 2 pairing mechanisms – legacy and Simple Secure Pairing (SSP). The interface is implemented in common/btauth.c.

7.1. Legacy pairing

The legacy pairing uses the following function when the controller asks the host to provide a PIN code:

- void bt_oem_get_pin_code(bt_bdaddr_t* bdaddr_remote);

When the stack calls bt_oem_get_pin_code() this function calls bt_hci_send_pin_code() with a hardcoded pin “0000”. The implementation may be more sophisticated. It can, for example, use different pin codes based on the address of the remote device. If the device has a display and keyboard it may show a UI to enter a PIN code first and call bt_hci_send_pin_code() later after the user has entered the PIN code. Also if support for legacy pairing is not desirable bt_oem_get_pin_code() may call bt_hci_reject_pin_code().

7.2. Secure Simple Pairing (SSP)

The SSP uses the following function:

- void bt_oem_ssp_callback(SSP_EVENT spp_event, void* event_param, void* init_param);

This function must implement a number of event handlers that may be sent during pairing. Our samples include an implementation of this function that implements “just works” association model (i.e., no PIN code entering) along with instructions on how to implement other models.

8. BT controller identification

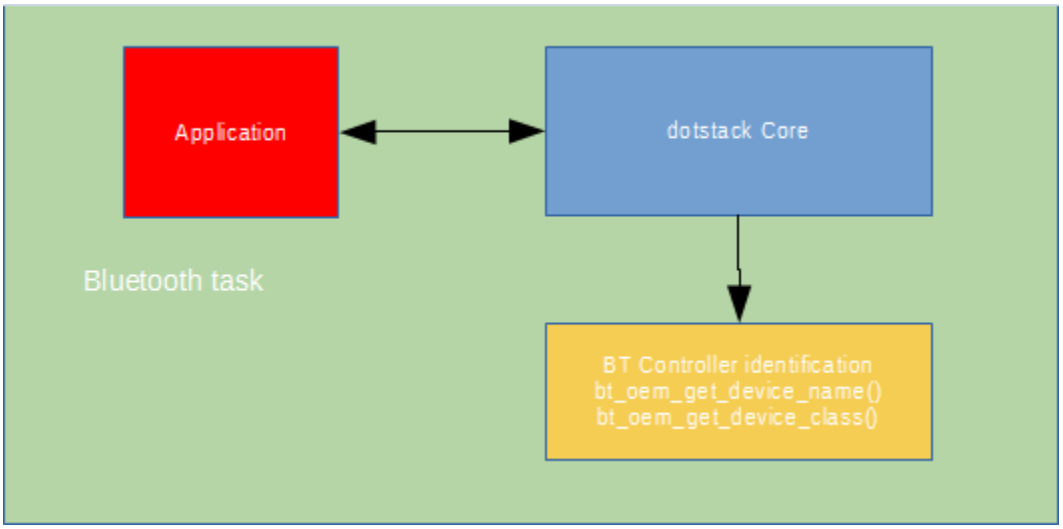

Figure 8

This interface is used to set the name and the class of device of the controller during stack initialization. The values set during initialization may later be changed with dotstack APIs. The interface consists of the following functions:

- const char* bt_oem_get_device_name(void);
- bt_long bt_oem_get_device_class(void);

These functions are usually defined in the main app file – the one that defines btapp_start().

8.1. Device name

bt_oem_get_device_name() must return a string no longer than 248 characters which will be written to the controller. Other devices will see this name when they discover the controller. The name of the device can be changed after initialization by calling bt_hci_write_local_name().

8.2. Class of device

bt_oem_get_device_class() must return the Class of device value which is a bitmask of values defined by the Bluetooth specification.