Dotstack Porting Guide.

dotstack Bluetooth stack is a C library and several external interfaces that needs to be implemented in the integration layer to run the stack on a concrete platform. The interfaces that needs to be implemented are:

- BT controller transport
- BT controller identification
- Authentication
- Timers
- Signal processing

The core library of the stack is not thread safe. Therefore, if the stack will run in a multi-threaded environment all callback function used by the above interfaces must be called from the thread that runs the stack. I.e., interrupt handlers cannot directly call callbacks. They have to use some kind of synchronization mechanism to notify the stack's task to call the callback for them. This applies to all other dotstack APIs. They have to be called from the BT thread.

BT controller transport

This interface allows the stack to communicate with the HCI controller. It 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);

The implementation of bt_oem_send() must send the specified number of bytes to the HCI controller and call the provided callback function.

The implementation of bt_oem_recv() must receive the specified number of bytes from the HCI controller and call the provided callback function.

The core library supports H4, 3-wire and BCSP host transports. All these transport work over UART. If USB or SDIO transport support is required, the easiest way to impmenet that is to emulate H4 transport.

BT controller identification

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

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

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().

bt_oem_get_device_class() must return the Class of device value which is a bitmask of values defined by the Bluetooth specification. The Class of device can be changed after initialization by calling bt_hci_send_cmd() (this function allows to send any HCI command to the controller).

Authentication

This interface is used when two controllers initiated a pairing procedure. There are 2 pairing mechanisms – legacy and Simple Secure Pairing (SSP). The legacy pairing uses the following function when the controller needs the host to provide it a PIN code:

- void bt_oem_get_pin_code(bt_bdaddr_t* bdaddr_remote);

In response to call of this function the integration layer must call either bt_hci_send_pin_code() or bt_hci_reject_pin_code(). These calls does not have to be made inside bt_oem_get_pin_code(). 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. The PIN code may also be hardcoded. In this case bt_oem_get_pin_code() may simply call bt_hci_send_pin_code(). If for some reason the device should not pair with the device that support only legacy pairing bt_hci_reject_pin_code() may be used to deny pairing requests from such devices. Remember, if UI runs in a different thread, after it's finished, the BT thread has to be notified (e.g., send a message to a queue) that it has to call bt_hci_send_pin_code() or bt_hci_reject_pin_code().

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 mode (i.e., no PIN code entering) along with instructions on how to implement other models. See btauth.c.

The result of pairing procedure is a link key which is used to authenticate the remote device and encrypt the traffic between devices. Link keys may be saved to a non-volatile storage in order to skip pairing after the device has been power cycled. To manager link keys there are 2 functions:

- void bt_oem_linkkey_notification(bt_linkkey_notification_t* lkn);
- void bt_oem_linkkey_request(bt_linkkey_request_t* lkr);

The stack calls bt_oem_linkkey_notification() when it receives a link key from the controller. The bt_linkkey_notification_t structure has the following definition:

```
typedef struct _bt_linkkey_notification_t
{
        bt_bdaddr_t bdaddr_remote;
        bt_linkkey_t key;
        bt_byte key_type;
} bt_linkkey_notification_t;
```

Where bdaddr_remote is the address of the paired remote device, key – the link key associated with the remote device, type – the type of the key. The integration layer may save this information to some sort of database that is stored in non-volatile memory. This will allow the key to be retrieved later when the stack calls bt_oem_linkkey_request().

The stack calls bt_oem_linkkey_request() when authentication has been initiated and the controller has asked the host to provide a link key for the remote device. bt_linkkey_request_t structure has the following definition:

```
typedef struct _bt_linkkey_request_t
{
        bt_bdaddr_t bdaddr_remote;
} bt_linkkey_request_t;
```

Where bdaddr_remote is the address of the remote device to authenticate. The integration layer has to look up a key for the device and if it is found call bt_hci_send_linkkey() passing the address of the device and the link key; otherwise - call bt_hci_send_linkkey() but pass only the address of the device. bt_hci_send_linkkey() has the following declaration:

- bt_bool bt_hci_send_linkkey(const bt_bdaddr_t* bdaddr_remote, const bt_linkkey_t* link_key, bt_hci_cmd_callback_fp cb);

The link_key parameter has to be NULL if there is no link key for the bdaddr_remote. Our examples include implementations of both functions that store link keys (depending on the platform) in on-chip flash, serial flash or file system. See btmgr.c.

Timers

Timers are used to implement various time outs. There are 2 functions that need to be implemented:

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

- void bt_oem_timer_clear(bt_uint timerId);

When the stack calls bt_oem_timer_set(), it must set the specified timer using any means available on the target platform. When the timer expires, the callback function passed to the bt_oem_timer_set() call must be called. The function must not wait until the timer expires. It must set the timer and exit immediately.

When the stack calls bt_oem_timer_clear(), the specified timer must be stopped. If the timer is already expired and a callback is currently pending, measures should be taken to cancel the callback.

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. Our samples include implementation that can provide all timers needed by the stack using one periodic timer of the target platform (e.g., SysTick on Cortex-M). See bttimer.c. Do not forget that timer callbacks has to be called from the BT thread.

Signal processing

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)
- bt_signal_process_pending()

The only purpose of this call is to let the BT thread that it needs to call bt_signal_process_pending() on the next iteration of the task's loop. bt_signal_process_pending() will go through the list of set signals and call their handlers. Here is how it implemented in FreeRTOS port:

```
void bt_oem_schedule_signals(void)

{

        bttask_setSignal(BTTASK_SIG_BTSIGNAL);

}

static void bluetoothTaskProc(void* params)
{
        bt_ulong signals;

        for (;;)
```

```
	{
		xSemaphoreTake(mSignalsSemaphore, portMAX_DELAY);

		taskENTER_CRITICAL();
		signals = mSignals;
		mSignals = 0;
		taskEXIT_CRITICAL();


		…

		if (signals & BTTASK_SIG_BTSIGNAL)
		{
			bt_signal_process_pending();
		}


		…
	}
}
```

Other interfaces

The above 5 interface are the only ones that need to be implemented in order to run the stack on any platform. If implemented correctly these will allow you to run any example that come with the stack but most likely with limited functionality. Many examples needs input from the user or display various information to the user, play audio etc. All such interface are not part of the stack. It is completely up to you how to implement them. The only thing to keep in mind that if the platform is multithreaded, the access to the dotstack API has to be synchronized. E.g., you have a thread that handles button presses and you want to send something to a remote device over SPP when the user presses a button. You cannot call bt_spp_send directly from the buttons handler thread. You have to, for example, send a message to a queue that the BT thread has access to. The BT thread receives the message and calls bt_spp_send.

One important interface is audio playback and recording. We do not have special interface for this. Our library only supplies audio data to the application which can handle it any way it likes. How the data is delivered to/from the application is profile specific. E.g., to receive audio stream through A2DP sink the app has to provide several buffers to the stack. When a packet is received the stack will fill a buffer and call an application callback. In this callback the app may process the data (decode it if it's encoded) and send to audio codec, or it may ignore what it receives. The data may be processed on the same thread or handed over to another thread. It is up to the application. However, our examples for some platforms (e.g., Cortex-M4) provide implementations of audio processing. They can be used as a starting point when porting the stack to another platform.