CASE STUDY

Good Connections

WaterRower Benefits from Bermondsey Electronics' services and its use of BELIeVE to ensure a Bluetooth-enabled product will work with all Android and Apple phones.

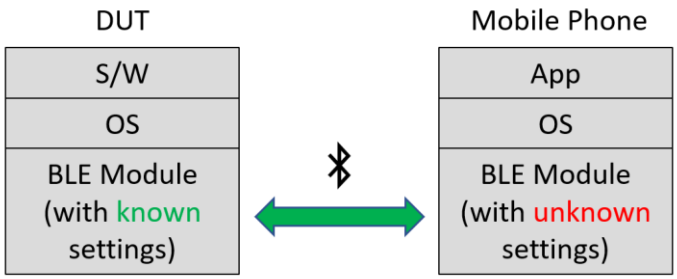
Developed by the Bluetooth Special Interest Group and introduced in 1998, Bluetooth is a short-range wireless technology used to transmit data over distances of typically less than 10 metres. An increasingly popular flavour of the technology is Bluetooth Low Energy (BLE). It was initially launched under the name Wibree in 2006 by Nokia but was subsequently integrated into Bluetooth (version 4.0) in 2009.

Those embedding BLE as a module into a product they are designing have control over the connection parameters, one of which is the connection interval (CI). It determines the bandwidth assigned during a session when both ends agree how frequently to transfer information.

Typically, the CI is set by the peripheral device. In most cases a mobile phone is the 'peripheral' device. For example, it might be added to the handsfree phone system of a car, which is the 'central' device.

Either device can request a change to the CI after it is set but it may not always be granted. This is because changing the CI can affect memory use and power consumption, the latter of which is important for battery-powered devices. Also, the OEMs of phones are often constrained by the radio implementation (of the BLE module vendor). This is normally opaque to the end user and transparent to the app developer, who also has no control over the parameter.

When designing a product that will pair with a mobile phone via BLE the onus is on the designer to accommodate the comms protocol of the phone. However, there is no way of knowing exactly how each phone manufacturer has implemented BLE. See figure 1.



Will the phone, with its unknown connection interval (CI) settings, allow the DUT to transfer enough data for the App to work?

Will CI change requests be accommodated? And if not, how might data transfer (bandwidth) be impacted? These questions must be answered… and were in the following case study.

Seeing the light

WaterRower is an OEM of indoor rowing machines. As suggested by the company's name, the user experience is similar to that experienced when rowing on water. The oarsperson's energy is used to move water within a drum. All machines are built (final assembly) in the USA, but the electronics are designed in the UK.

Bermondsey Electronics first supported WaterRower in 2018, initially to help with the integration of FreeRTOS and some complicated software stacks into a product under development. Since then, Bermondsey has taken on greater responsibilities.

In 2022, Bermondsey was developing a BLE-enabled accessory to work with all mobile devices (with Android or iOS operating systems). The accessory communicates with a mobile device but needs a minimum bandwidth to be responsive.

An early version of the product worked fine when transferring data with a desktop PC via Bluetooth. It was not known how well the product would work when connecting to a mobile phone, particularly one with limited flexibility where accommodating changes to the CI is concerned.

Without access to all the various makes and models of phones with which the rowing machine was to interface, it was decided to plan for a worst-case comms scenario. But that raised another problem. How much time should be spent on coding in light of not really knowing what that worst-case scenario was?

Bermondsey decided to simulate (on a desktop) the communications taking place. For this the Bermondsey Electronics Limited Integration Verification Engine (BELIeVE) was used to control a Nordic Semiconductor BLE driver – see figure 2.

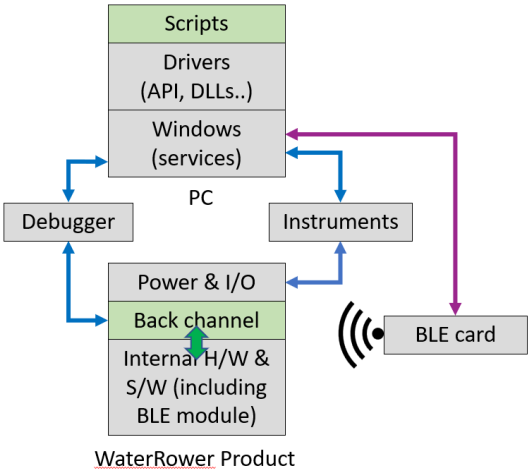


Figure 2. Above, the set up for talking to WaterRower's product. The 'Scripts' are written and executed on BELIeVE.

BELIeVE is a JavaScript-based application that runs on a PC (with Windows 10 or higher). It automates the operation of networkable instrumentation to exercise a device under test (DUT) and verify that its hardware and software are functioning together as intended.

BELIeVE's DLLs are written in C# so could not directly control the Nordic Semiconductor BLE driver. The solution was to create an API to run on the BLE driver. Bermondsey developed the API using ZeroMQ (zmq) to create push/pull sockets to and from the target. This new zmq application transfers data in and out of Python; where the Python application is freely available from Nordic Semiconductor.

Using the API, Bermondsey was able to control an update_connection_interval function available within the Nordic Python driver. Here is the code:

```
@nrf_error_handler
def update_connection_interval(self, min_conn_interval_ms: float,
max_conn_interval_ms: float):
    if self.active_conn_handle is None:
        raise NordicSemiException   # This forces the error handler to return FAIL
    else:
```

```
        conn_params = self.adapter.driver.conn_params_setup()  # Get default values
        conn_params.min_conn_interval_ms = min_conn_interval_ms  # Overwrite values of
interest
        conn_params.max_conn_interval_ms = max_conn_interval_ms
        self.adapter.conn_param_update(self.active_conn_handle, conn_params)
```

The `@nrf_error_handler` decorator simply returns a value if the function fails or if its exception handler is called. The function definition simply calls the Python method. Note: there is little overhead in doing this since Python will have to locate this method anyway.

An update_connection_interval function was now available through BELIeVE. Ditto an error_handler function which simply returns and an error code (i.e., a value other than NRF_SUCCESS) back up the call chain.

BELIeVE was now able to call both functions. In BELIeVE's DLLs the code was this:

```
    public static ERR_CODE update_connection_interval(float min_conn_interval_ms,
float max_conn_interval_ms)
    {

ZmqClient.send_ble_command($"{rc_driver}.update_connection_interval({min_conn_interval
_ms}, {max_conn_interval_ms})");
        return error_handler();
    }
```

The parameters expressed – min_conn_interval_ms and max_conn_interval_ms – are the literal values passed the Nordic device. The other end may then reply appropriately, either choosing a value in this range or declining the request.

Bermondsey used this function in the BLE class to hide the internal ZmqClient instance from the rest of the DLL. Finally, at the top level of the DLL, Bermondsey exposed a method to invoke the functionality.

```
        public ERR_CODE UpdateConnectionInterval(float min_conn_interval_ms, float
max_conn_interval_ms)
        {
            return RemoteBleDriver.update_connection_interval(min_conn_interval_ms,
max_conn_interval_ms);
```

```
}
```

Note: by hiding the internal methods and classes the DLL uses it will be easy to make changes in the future; if Nordic supply a change to the API, for example, in which case the public API does not necessarily have to change.

In BELIeVE

With all the above in place, Bermondsey wrote a test script to run in BELIeVE. The script performs two tests, one with a long CI and one with a short, to demonstrate the change in bandwidth that could be achieved.

Here is the test script. Firstly, declarations are made:

```
var testString = "0, 1, 2, 3,";
var testData = "";
var loopSize = 5;
var timer = 0;
var counter = 0;
const transferCount = 10;
const oldMin = 100;
const oldMax = 150;
const newMin = 7.5;
const newMax = 20;
```

A counter is created

```
for(; counter < loopSize; counter++)
    {
      testData += testString;
    }
```

The following code establishes a connection to the target device and the CI is set to be between 100 and 150ms. This requests the other end of the link to be available at that interval between connection events.

```
testData = "[" + testData.slice(0, testData.length - 2) + "]";
DLL_TOOL.LogMessage("Length of data packet = " + (4*loopSize), 10, 0);
```

```
e = DLL_BLE.StartController("NRF52",MSVAR_BLE_COM_Port, TGVAR_ZMQ_TIMEOUT);
DLL_BLE.ErrorHandler(e);

DLL_TOOL.LogMessage("Connecting to device...", 0, 0);
e = DLL_BLE.ConnectToDevice(MSVAR_TGT_DEVICE_NAME, TGVAR_BLE_TIMEOUT);
DLL_BLE.ErrorHandler(e);

DLL_TOOL.LogMessage("Set connection interval min = " + oldMin + "ms, max = " + oldMax, 10, 0);
DLL_BLE.UpdateConnectionInterval(oldMin, oldMax);

DLL_TOOL.StartStopwatch();
```

Next, we loop over the test data that we want to send and transfer it into the DUT. By transferring a consistent amount of data and measuring the transfer time, we can work out the bandwidth this CI offers. We will repeat this later with changed CI settings, to see how the bandwidth is affected by the changes.

```
for(loopSize = transferCount; loopSize > 0; loopSize --)
{
   DLL_BLE.WriteCharacteristic(testData, MSVAR_TGT_SERV_UUID, MSVAR_TGT_CHAR_UUID);
}

timer = DLL_TOOL.GetStopwatchMSec();
```

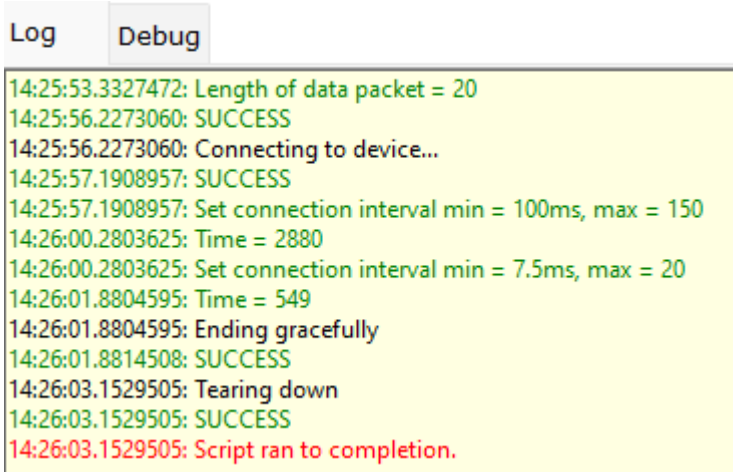
Next, the CI interval is given new min and max limits.

```
DLL_TOOL.LogMessage("Time = " + timer, 10, 0);
DLL_TOOL.LogMessage("Set connection interval min = " + newMin + "ms, max = " + newMax, 10, 0);
DLL_BLE.UpdateConnectionInterval(newMin, newMax);

DLL_TOOL.StartStopwatch();

for(loopSize = transferCount; loopSize > 0; loopSize --)
```

```
{
   DLL_BLE.WriteCharacteristic(testData, MSVAR_TGT_SERV_UUID, MSVAR_TGT_CHAR_UUID);
}

timer = DLL_TOOL.GetStopwatchMSec();

DLL_TOOL.LogMessage("Time = " + timer, 10, 0);

DLL_TOOL.LogMessage("Ending gracefully", 8, 0);
```

Results

Here are the result of the above code executing:



The opposite end of this link (a Nordic development kit emulating a device under test) selected a connection interval at the top end of the offerings, i.e. 150msec for the earlier setting and 20msec for the later setting.

Conclusion

Thanks to being able to control the CI it was possible to simulate the BLE communications with any mobile phone or tablet. Bermondsey Electronics knew how much data needed to be transferred so, even if a phone would only allow an upper CI limit (to preserve battery life) the software in the WaterRower accessory could be optimised accordingly. Moreover, it could be shown to work – essential for product launch – even without access to the phone.