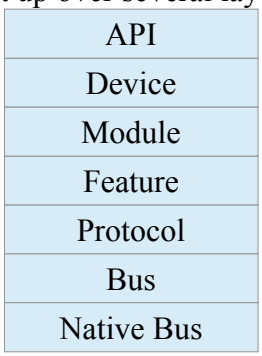
The SeaBreeze architecture is built up over several layers. These are shown below.



Native Bus

The native bus layer is responsible for providing a read/write data interface to the native hardware. In OmniDriver, this delegates to the UniUSB and UniRS232 projects which provide native bus access through Java Native Interface (JNI) on each supported platform. Bus-specific functionality, such as device enumeration, baud rates, etc. are handled at this level. If the bus allows devices to be probed automatically, then it can provide a list of device locations that can be used to open specific devices.

Bus

The bus layer is intended to abstract away data transfers to simple send and receive semantics. This is assisted using TransferHelpers. While the bus itself represents the physical medium that will carry data, there are often some details that are required to know how to use that bus for a particular data transfer. For instance, in USB, the endpoints must be known for each send and receive operation, and in Ethernet, a socket connection to a particular port may be required.

TransferHelpers represent a particular way of sending data over a particular bus, encapsulating the details (like USB endpoints) into an opaque object that simply offers to send and receive bytes. Before a transfer can occur, the bus is asked to provide an appropriate TransferHelper, and the TransferHelper is then used to execute the transfer. The bus can differentiate between potentially many TransferHelpers to decide which one to use for a given transfer based on a provided ProtocolHint.

A good example of how this works is provided by the USB2000. This device has a few peculiarities that have to be worked around in the driver. There are two basic kinds of transactions with the device: control and spectrum. Control transfers go out on Endpoint 2 OUT and come back on Endpoint 7 IN. Spectrum requests also go out on Endpoint 2 OUT, but the spectrum is returned on Endpoint 2 IN. The assignment of endpoints to these roles is handled within the TransferHelper: when a transfer needs to take place, an appropriate hint is given (control or spectrum), and the Bus provides a TransferHelper that already knows which endpoints to use.

A deeper example is the USB4000. In this device, Endpoint 1 OUT is used for control requests and spectrum requests. Control responses (e.g. EEPROM values, FPGA register contents) are returned on Endpoint 1 IN. Spectra are returned on Endpoint 2 IN if Full-Speed USB is being used, and a combination of Endpoints 2 and 6 IN if High-Speed USB is being used. To cover the cases, three TransferHelpers exist: one for control messages, one for spectra under Full-Speed, and one for spectra under High-Speed. Since the High-Speed case splits the data stream across multiple endpoints, the TransferHelper also reassembles the message into a single contiguous stream so that the next layer

above can parse it without any concern about these bus-specific machinations. When the USB4000 is first opened, a test is performed to identify whether USB Full-Speed or High-Speed is being used, and the correct TransferHelpers are created and cached accordingly.

TransferHelpers can assist in other ways. On the Jaz, the same protocol (for the messaging protocol) can be used over both USB and Ethernet. However, in the USB case, all messages being sent to the device must have two bytes prepended ahead of the message: the value 0x10, and then a one-byte length of the remainder of the message. The JazUSBMessagingTransferHelper inserts these extra bytes into messages that are destined for USB, while the TransferHelper for Ethernet simply sends the commands on without modification. In the IDRaman driver, the device expects newlines to be sent after every command and it sends command prompts after each response (intended for a terminal program), so the IDRamanRS232TransferHelper (which extends the TerminalTransferHelper) automatically adds newlines and strips off the command prompts so that the protocol parser can focus on the content of the message.

Protocol

The protocol layer is responsible for all encoding and decoding of byte streams. It is intended to capture each kind of message that can be sent to or received from a device. Since device protocols themselves can be hugely varied, this does not make many assumptions other than these: a “protocol implementation” must implement an interface that allows certain types of messages to be sent or received, and it must be able to do so with the help of an appropriate TransferHelper to send and receive the bytes.

It is important, when working at the Protocol layer, to focus only on how particular commands or queries might be encoded, and how responses should be decoded. For instance, bounds-checking in the Protocol layer should be limited to only those cases where the protocol itself imposes the bound (for instance, if only a single lamp bulb can be controlled the way the protocol is defined, then it may throw an error if an attempt to control a higher-index bulb is made). On the other hand, the Protocol layer should not care what the full range of values accepted by the device might be beyond how it would encode them (e.g. within the size of fields specified in the device's user manual).

Protocol implementations sometimes are referred to as “clean”, which generally means that there is a one-to-one mapping from Feature-layer requests to protocol messages. In this case, the protocol layer implementation is minimal, simply choosing the correct message to send and sending it. Protocol implementations can also be much more complex, arranging a number of transactions with a device to achieve the requirement of the protocol interface. The difference between these will depend entirely on the device having its protocol implemented.

Unfortunately, protocol implementations are one of the most time-consuming parts of adding a new product to the driver. This is due simply to the need to very carefully represent every possible input and output from the device that might be encountered. The good news is that these Protocol implementations are often easy to reuse if a similar device is added.

The protocol interface at the top end of the protocol layer provides a set of functionality that is implemented with a set of protocol messages. These interfaces are intended to be implemented for potentially many device command sets. For instance, for a single device (e.g. USB2000+), there could be two implementations that provide access to the spectrometer: the legacy USB protocol, and the legacy RS232 interface. If the protocol implementations for both of these exist, then the same spectrometer feature (in the next layer up) could use either one, depending on which bus happened to be connected for that session.