Tutorial

Getting started

- Clone the workshop repository.
- Run `setup.cmd` from the `./bonsai` directory to install Bonsai and its dependencies.

Following the examples

Each example builds on the previous one, so it is recommended to follow them in the order presented in the table of contents.

If you run into problems assembling the examples, you can copy-and-paste each snippet by clicking the clipboard icon (top-right corner) of each code block, and pasting it into the Bonsai workflow editor.

If you have any questions or find any issues, please open an Issue on the workshop repository

More documentation

Community

- Q&A, community, forum
- Bonsai documentation

Spec

- Harp Protocol
- Harp Device
- Device technical references

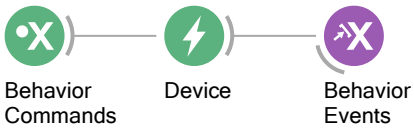
Tutorials

- Using the Bonsai.Harp packages
- Python data interface
- AIND Harp devices
- Using Harp.HobGoblin

Connecting to the Harp device

- Add the `Device(Harp.Behavior)` operator and assign the `PortName` property.
- Add a `PublishSubject` operator and name it `BehaviorEvents`.
- Add a `BehaviorSubject` source, and name it `BehaviorCommands`. A Source Subject of a given type can be added by right-clicking an operator of that type (e.g.`Device`) and selecting `Create Source -> BehaviorSubject`.
- Run Bonsai and check check the output from the device.

> **TIP**
>
> Any operator in Bonsai can be inspected during runtime by double-clicking on the corresponding node. This will display the output of the operator in a floating window.



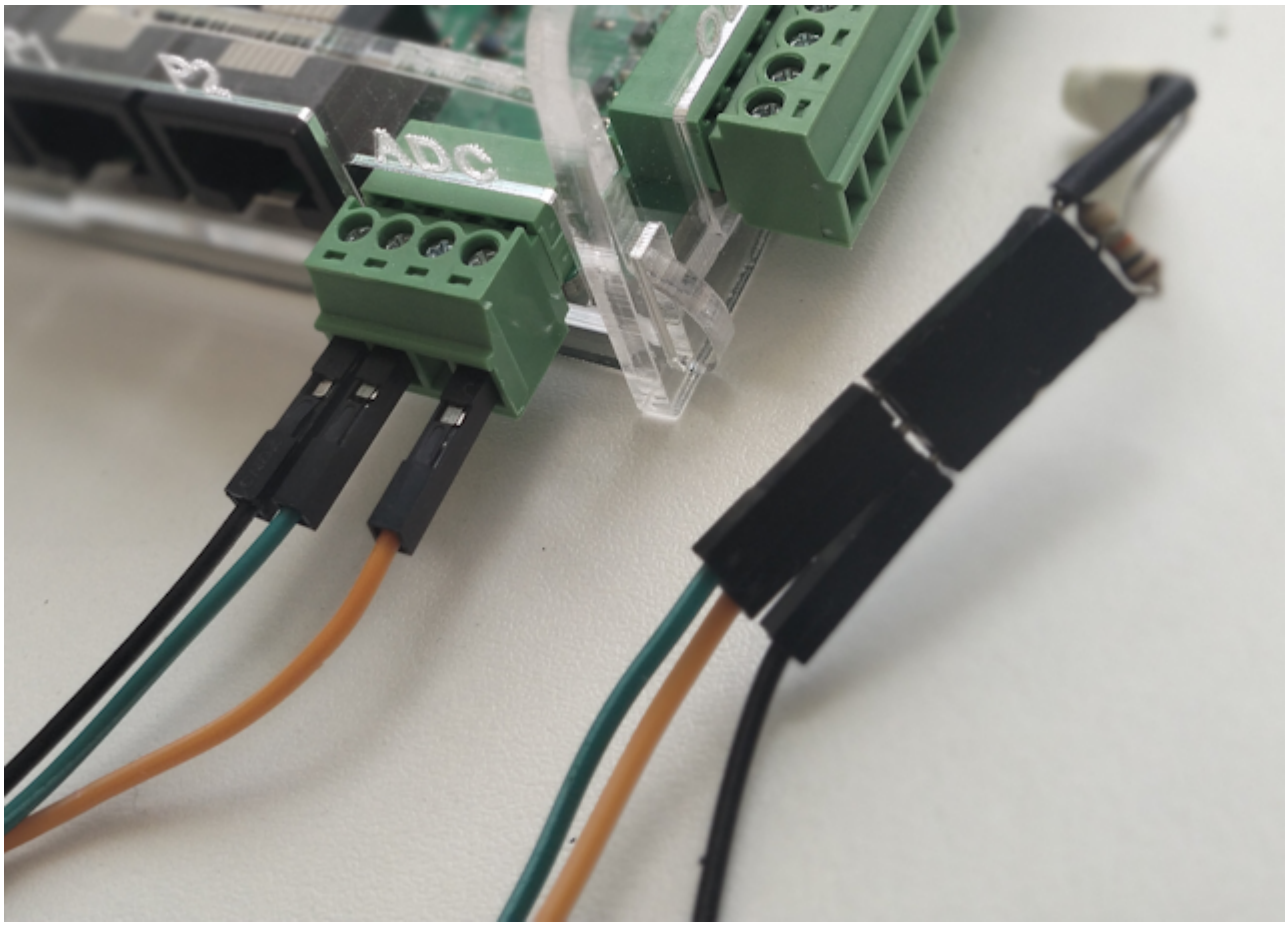
> **NOTE**
>
> Using the device-specific `Device` operator is the recommended way to connect to a Harp device. This operator runs an additional validation step that ensures that the device you are attempting to connect to matches the interface you are trying to use. For cases where this check is not necessary, you can use the generic `Device` operator, which is available in the `Bonsai.Harp` package.

Filtering Harp messages

- As you probably noticed right after running the previous snippet, the device is sending a lot of messages. This is because this specific board has a high-frequency periodic event associated with ADC readings. We will come back to this point later, but for now, we will filter out these messages so we can look at other, lower-frequency messages from the device.
- To filter messages from a specific register we can use `FilterRegister(Harp.Behavior)` operator. This operator can be added in front of any stream of Harp messages in the workflow.
- Add a `SubscribeSubject` and subscribe to the `BehaviorEvents` stream.
- Add the `FilterRegister(Harp.Behavior)` operator and assign the `Register` property to the register you want to filter on (`AnalogData`).
- Modify the `FilterType` property to `Exclude` to filter out the messages from the specified register.
- Check the output of `FilterRegister`

Parsing AnalogData `Event` messages

Build the following circuit before start:

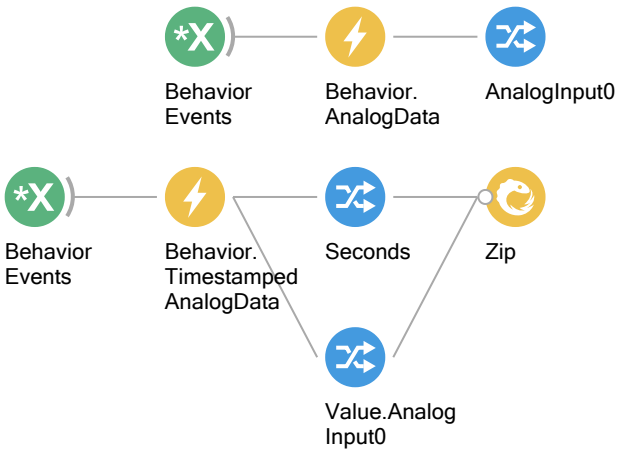


In a previous exampled we mentioned referred to `AnalogData` as a high-frequency event that carries the ADC readings. It is important to note that, as opposed to `FirmwareVersionHigh` which belongs to the core registers common across all Harp devices, `AnalogData` is a Harp Behavior specific register. As result, we must use the `Harp.Behavior` package to parse this register:

- Subscribe to the `BehaviorEvents` stream.
- Add a `Parse(Harp.Behavior)` operator
- Set `Register` to `AnalogData`
- The output type of `Parse` will now change to a structure with the fields packed in this register.
- To select the data from channel 0, right-click on the `Parse` operator and select `AnalogInput0`.
- Run Bonsai and check the output of the `AnalogInput0` stream by double-clicking the node.

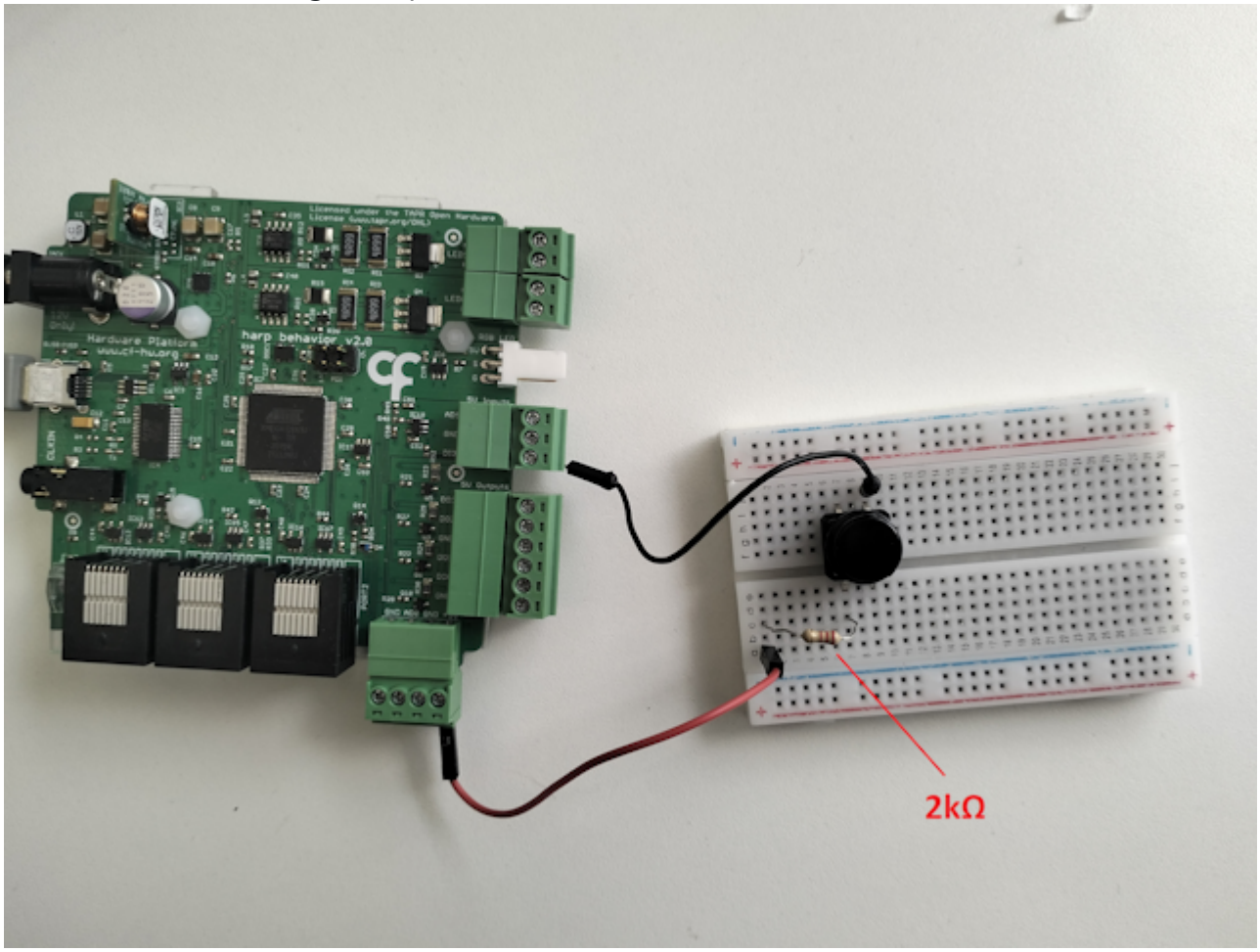
You will notice that despite the timestamp being present in the message, the `AnalogInput0` output stream is not timestamped. This is because the `Parse` operator does not propagate the timestamp from the original message by default. In cases where the timestamp is necessary, for each `<Payload>` we have a corresponding `Timestamped<Payload>` that can be selected in all `Parse` operators. This will add an extra field to the parsed structure, `Seconds`, that contains the timestamp of the original message (in seconds):

- Modify the `Register` property to `TimestampedAnalogData`
- Select the `AnalogInput0` and `Seconds` members from the output structure.
- Optionally pair the elements into a `Tuple` using the `Zip` operator.



Parsing a DigitalInput `Events`

Assemble the following example:



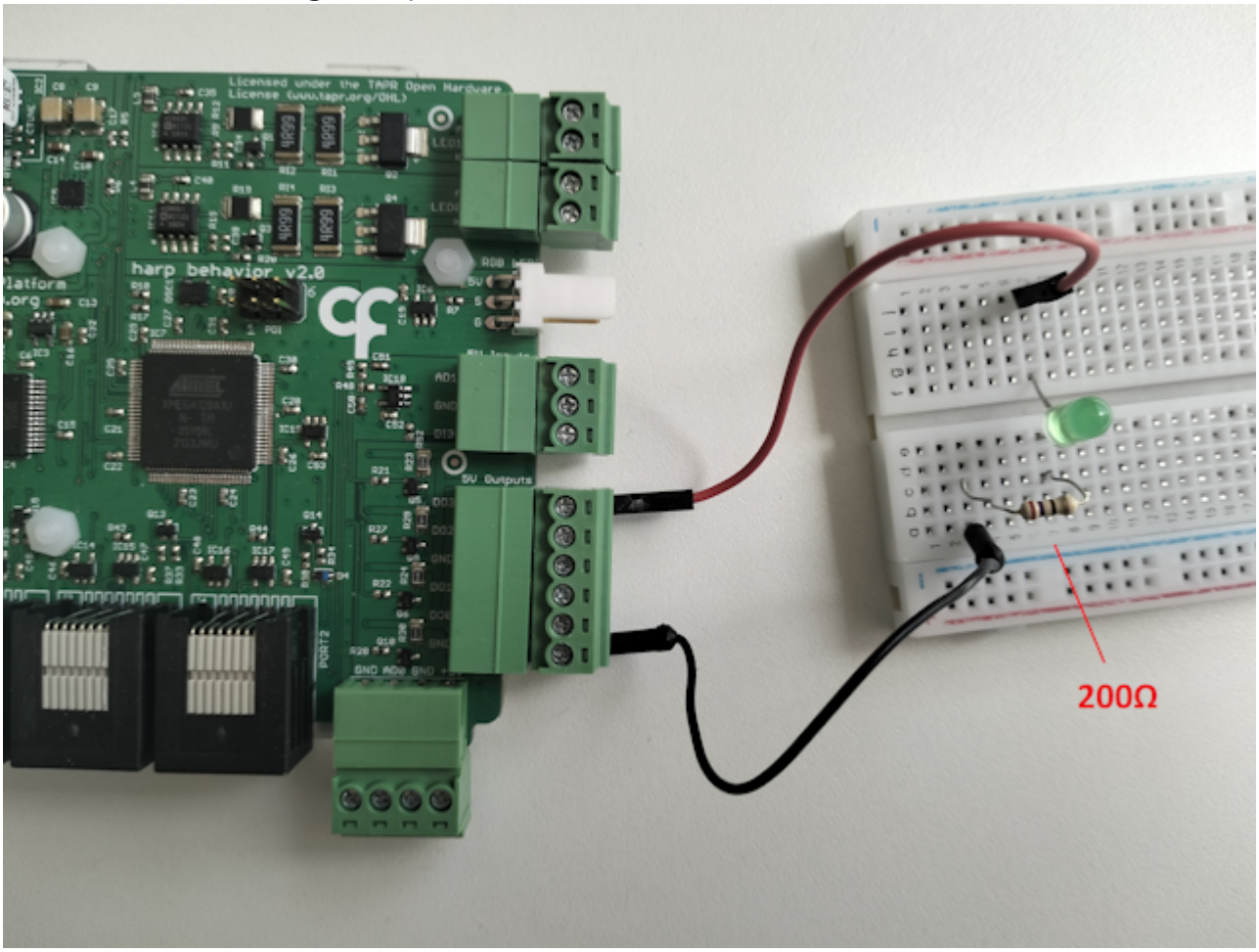
While the `AnalogData` is a register that sends periodic message (~1kHz), other messages are triggered by non-period events. One example is data from the digital input lines. In the Harp Behavior board, register `DigitalInputState` emits an event when any of the digital input lines change state. It is important to note that similar to other devices (e.g. Open-Ephys acquisition boards), the state of all lines is multiplexed into a single integer (`U8`), where each bit represents the state (1/0) of each line. As a result, depending on the exact transformation you want to apply to the data, you may need to use the `Bitwise` operators to extract the state of each individual line:

- Subscribe to the `BehaviorEvents` stream.
- Add a `Parse(Harp.Behavior)` operator
- Set `Register` to `DigitalInputStatePayload` (You can also use `TimestampedDigitalInputState` if you need the timestamp)
- To extract the state of a specific line, use the `HasFlag` operator and set `Value` to the line you want to extract (e.g. `DI3`).

- Because the state of `DigitalInputState` changes when ANY of the lines change, we tend to use the `DistinctUntilChanged` to only propagate the message if the state of the line of interest changes.
- Finally, to trigger a certain behavior on a specific edge, we add a `Condition` operator to only allow `True` values to pass through. The behavior can easily be inverted by adding a `BitWiseNot` operator before, or inside, the condition operator.

Sending `Commands` to the device

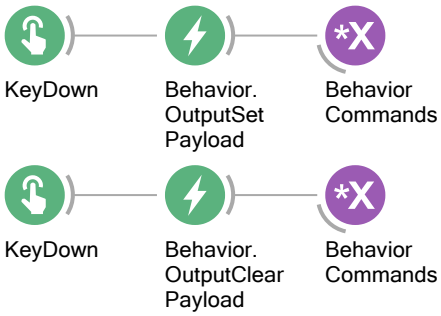
Assemble the following example:



Change the state of the digital output line

The Harp Behavior device has a set of four registers that can be used to control the state of the digital output lines: `OutputSet`, `OutputClear`, `OutputToggle` and `OutputState`. For simplicity, we will only use the `OutputSet` and `OutputClear` registers in this example. These registers are used to set or clear the state of a specific line, respectively. Similarly to the `DigitalInputState`, the value of this register also multiplexes the value of all the lines. First, we will set the state of line `DO3` to `High`:

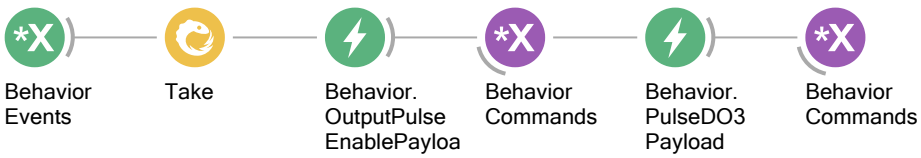
- Add a `KeyDown(Windows.Input)` operator and set the `Filter` property to a specific key (e.g. `1`).
- Add a `CreateMessage(Harp.Behavior)` operator in after the `KeyDown` operator.
- Select `OutputSetPayload` under `Payload`. Make sure the `MessageType` is set to `Write` since we will now be asking the device to change the value of one of its registers.
- In the property `OutputSet`, select the line you want to turn on (e.g. `DO3`).
- Replicate the previous steps to clear (turn off) the state of the line `DO3` by using the `OutputClearPayload` instead, and the `KeyDown` operator with a different key (e.g. `2`).
- Verify that you can turn On and Off the line `DO3` by pressing the keys `1` and `2`, respectively.



Changing the pulse mode of a digital output line

In most Harp devices you will find registers dedicated for configuration rather than "direct control". One example is the `OutputPulseEnable` register in the Harp Behavior board. This register is used when the user wants to pulse the line for a specific, pre-programmed, duration (e.g. opening a solenoid valve for exactly 10ms). To use this feature:

- Subscribe to the `BehaviorEvents` stream.
- Add a `Take` operator.
- Add `CreateMessage(Harp.Behavior)` operator in after the `Take` operator.
- Select `OutputPulseEnablePayload` under `Payload`. Make sure the `MessageType` is set to `Write`.
- Select the line you want to pulse (e.g. `DO3`), and add a `MulticastSubject` operator to send the message to the device.
- Add another `CreateMessage(Harp.Behavior)` operator after the `MulticastSubject` operator.
- Select `Pulse<Pin>Payload`, and set the value to the number of milliseconds you want this line to be high for on each pulse.
- Add a `MulticastSubject` operator to send the message to the device.
- Verify you see a pulse on the line `DO3` every time you press the key `1`.

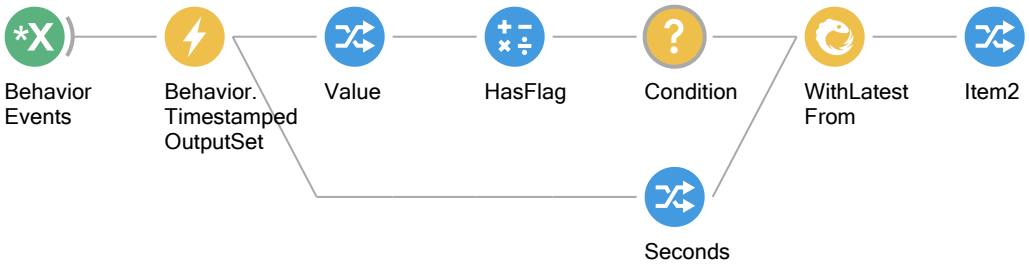


> **NOTE**
>
> The `BehaviorEvents->Take(1)` pattern will wait for the first message from the device before sending any commands, guaranteeing that the device is ready to receive commands.

Getting the timestamp of a Write message

While we know that the state of the line `DO3` is changing, we do not have access to WHEN this change is occurring. Remember that for each `Write` message issued by the computer as a command, a `Write` message reply should be sent back from the device. To grab the timestamp of the reply message:

- Subscribe to the `BehaviorEvents` stream.
- Add a `Parse(Harp.Behavior)` operator and set the `Register` to `TimestampedOutputSet`.
- Expose the `Value` and `Seconds` members of the output structure.
- Add a `HasValue(DO3)` after `Value` to extract the state of the line `DO3`.
- Add a `Condition` operator to only allow `True` values to be propagated.
- Recover the initial timestamp of the message by using a `WithLatestFrom` operator connecting the output of `Condition` and `Seconds`.



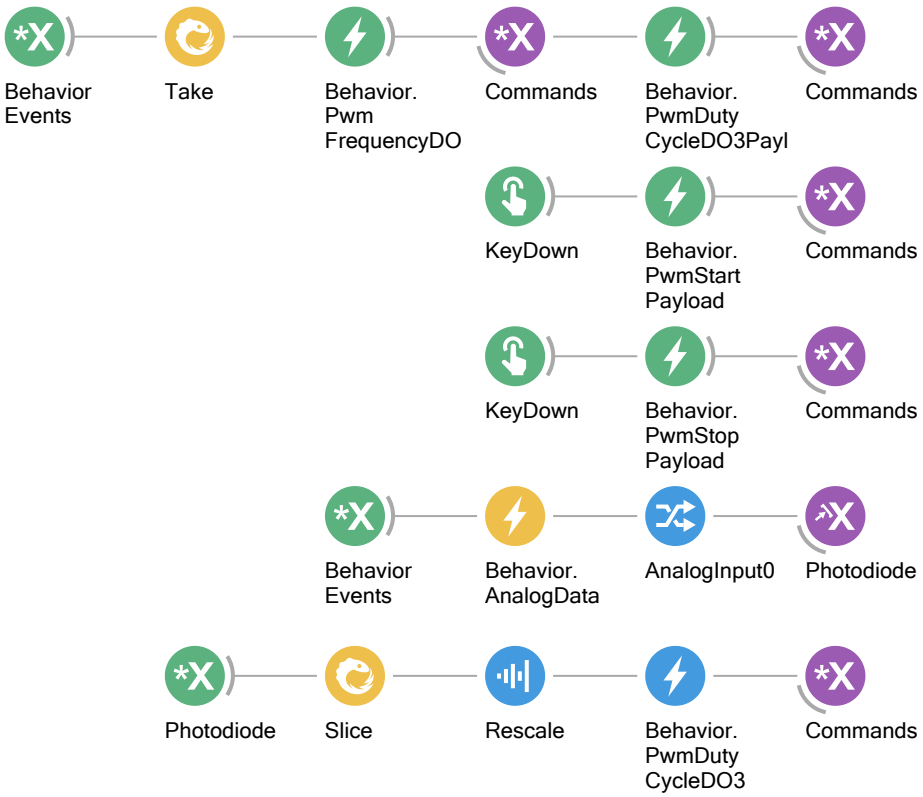
> **NOTE**
>
> More documentation on how to manipulate timestamped messages can be found here

Closing the loop with PWM

Building on top of the Analog Data section, this example will walk you through how to achieve "close-loop" control between the duty-cycle of a closed-loop signal and the value of an ADC channel. This example also highlights one of the major advantages of having a computer in the loop: the ability to easily change the behavior of the system by changing the software.

- Configure `DO3` to be a PWM output by replicating the previous sections but instead of using the `Pulse<Pin>Payload`, configure the initial frequency (e.g. 500Hz) and duty cycle (e.g. 50%) of the PWM by using `PwmFrequency<Pin>Payload` and `PwmDutyCycle<Pin>Payload`.
- Add a `KeyDown(Windows.Input)` operator and set the `Filter` property to a specific key (e.g. `Up`).
- Add a `CreateMessage(Harp.Behavior)` operator in after the `KeyDown` operator, and set it to `PwmStart` and match the value to the pin you are using (e.g. `DO3`).
- Repeat the previous steps but now set the `PwmStop` register to stop the PWM signal when the key `Down` is pressed.

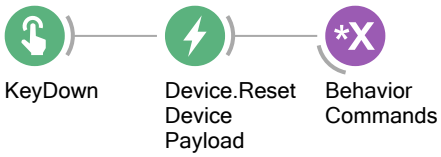
- Verify that you can start and stop the PWM signal.
- Resume the pattern in from the Analog Data section. and publish the value of the ADC channel 0 via a `PublishSubject` named `Photodiode`.
- Add a `Slice` operator to down-sample the signal to a more manageable update frequency (e.g. 100Hz) by setting the `Step` property to `10`. This is advised since the Behavior board is only spec'ed to run commands at 1kHz. Different hardware / functionality may require different sampling rates, so be sure to run tests before deploying the system.
- Subscribe to the `Photodiode` stream and add a `Rescale` operator. According to the documentation of the Harp Behavior board, the duty cycle register only accepts values between 1 and 99. As a result, we need to rescale the value of the ADC channel to match this range. Set the `Max` and `Min` properties to the maximum and minimum values of the Photodiode signal. Set `RangeMax` and `RangeMin` to 99 and 1, respectively. Finally, to ensure values are "clipped" to the range, set `RescaleType` to `Clamp`.
- Finally, add a `Format(Harp.Behavior)` operator after the `Rescale` node. `Format`, similarly to `CreateMessage` is a Harp message constructor. It differs from `CreateMessage` in that it uses the incoming sequence (in this case the rescaled value of the ADC channel) to populate the message, instead of setting it as a property.
- Add a `MulticastSubject` operator to send the message to the device.



Resetting the device

In some cases, you may want to reset the device to its initial known state. The Harp protocol defines a core register that can be used to achieve this behavior:

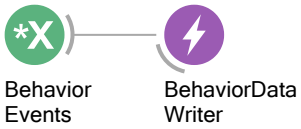
- Add a `KeyDown(Windows.Input)` operator and set the `Filter` property to a specific key (e.g. `R`).
- Add a `CreateMessage(Bonsai.Harp)` operator in after the `KeyDown` operator.
- Select `ResetDevicePayload` in `Payload`, and `RestoreDefault` as the value of the payload.
- Add a `MulticastSubject` operator to send the message to the device.
- Run Bonsai. The board's led should briefly flash to indicate that the reset was successful.



Logging

Logging messages from device

- Subscribe to `BehaviorEvents`
- Add a `Device.DataWriter(Harp.Behavior)` operator
- Set the `Path` property to the folder where you want to save the data (e.g. `./data/MyDevice.harp`)



Data Interface

Setting up the python environment

To analyze the data, you will need to install harp-python package.

```
python -m venv .venv
.venv\Scripts\activate
pip install harp-python
```

```
device = harp.create_reader("./data/MyDevice.harp")
data = device.DigitalInputState.read()
data_analog = device.AnalogData.read()

plt.figure()
plt.plot(data)
plt.plot(data_analog)
plt.xlabel("Harp time (s)")
plt.show()
```