

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

- [Harp Protocol](#)
- [Harp Device](#)
- [Using the Bonsai.Harp packages](#)
- [Device technical references](#)
- [Python data interface](#)
- [AIND Harp devices](#)
- [Bonsai documentation](#)
- [Q&A, community, forum](#)

Workshop Kit Components

The following components are used in to assemble the circuit examples used in the workshop:

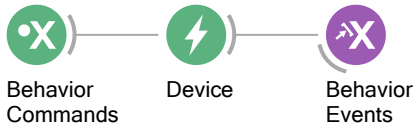
- 4x, Wago In-Line Splice Connector, 221-2401, [link](#)
- 1x, 10K Ohm resistor, RNF14FTD10K0, [link](#)
- 2x, 330 Ohm resistor, RNMF14FTC330R, [link](#)
- 1x, Red/Green bidirectional LED, 5219459F, [link](#)
- 1x, Photoresistor, NSL-6910, [link](#)
- 1x, Hall Effect Door Sensor, 59135-030, [link](#)
- 1x, Door Sensor Magnet, 57135-000, [link](#)

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 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`

NOTE

Sometimes it may be easier to exclude registers using the generic API rather the device-specific one. This can be done using the `FilterRegister(Bonsai.Harp)` operator from the `Harp` package. This operator allows you to filter messages based on the register address number (e.g. `Address=44`), but it is otherwise interchangeable with the previous operator.



Behavior
Events



Behavior.
AnalogData



Behavior
Events



FilterRegister

Core Registers

Reading the firmware version using the low-level API

- Each Harp device has a core register that contains the firmware version flashed on the board. An easy way to check what is the Major version of the firmware is to read from this register. [This register follows the following structure](#):

FirmwareVersionHigh:

address: 6
type: U8
access: Read

To read from this register, we need to create a **Read**-type message to this register:

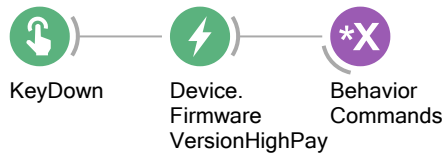
- Add a **CreateMessage(Bonsai.Harp)** operator
- Select **CreateMessagePayload** under **Payload**. This will allow us to specify all the fields of the Harp message.
- Populate the properties **MessageType**, **Address** and **PayloadType** with the register information above.
- In several cases we will want to trigger the reading of the register with some other event. For debugging, one useful trick is to use **KeyDown(Windows.Input)** operator that sends a notification when a key is pressed. To prevent sporadic triggers set the **Filter** property to a specific key (e.g. **1**).
- Connect this operator to the **CreateMessage** operator.
- In 3, we used **SubjectSubject** to create a "one-to-many" pattern (i.e. one **Device** source to two parallel **FilterMessage** operators). When sending commands to a device we usually want to create a "many-to-one" pattern instead. This can be done by using the **MulticastSubject** operator with the **BehaviorCommands** as the target subject.
- Add a **MulticastSubject** operator after the **CreateMessage** operator.
- Run Bonsai and click **1** to trigger the reading of the firmware version. What do you see in the filtered device output?



Reading the firmware version using the abstracted API

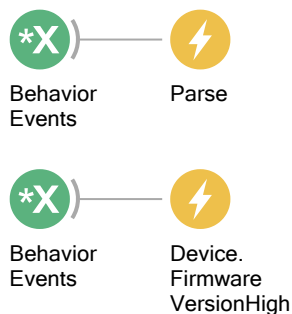
- The ability to manipulate "raw" Harp messages is very useful for debugging new devices. However, for most applications, we will want to use the abstracted API instead of having to know the register specification as in the previous point:
- Add a **CreateMessage(Bonsai.Harp)** operator

- Select **FirmwareVersionHigh** under **Payload**. This change will automatically populate the **Address** and **PayloadType** to match the select register. You will still need to assign a **MessageType**, in this case, **Read**.
- Re-run the previous example using this operator instead.



Parsing the message payload

- After the last step, you should see a message from the register **FirmwareVersionHigh**. However, we have yet to parse the message payload to see the actual firmware version.
- Replicating the previous steps, we will start by learning how to parse the payload using the low-level API:
 - Add a **Parse(Bonsai.Harp)**. This operator will not only parse the Harp Message payload to the specified type but also filter out messages that do not match the specified parsing pattern (e.g. other registers).
 - Assign the properties using the same values from the previous example.
- Once again, we can also use the abstracted API to simplify the parsing process:
 - Add a **Parse(Bonsai.Harp)** operator
 - Select **FirmwareVersionHigh** under **Payload**
 - Re-run the previous example using this operator instead.



Parsing AnalogData Event messages

Build the following circuit before start:

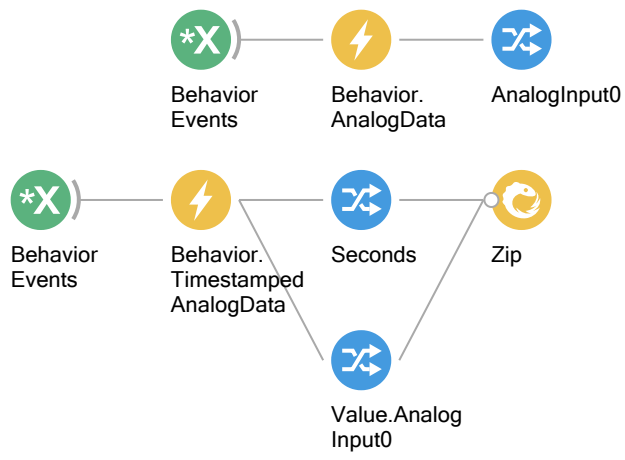


In a previous example we mentioned referred to **AnalogData** as a high-frequency event that carries the ADC readings. It is important to note that, as opposed to **FirmwareVersionHigh** which belongs to the core registers common across all Harp devices, **AnalogData** is a Harp Behavior specific register. 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.

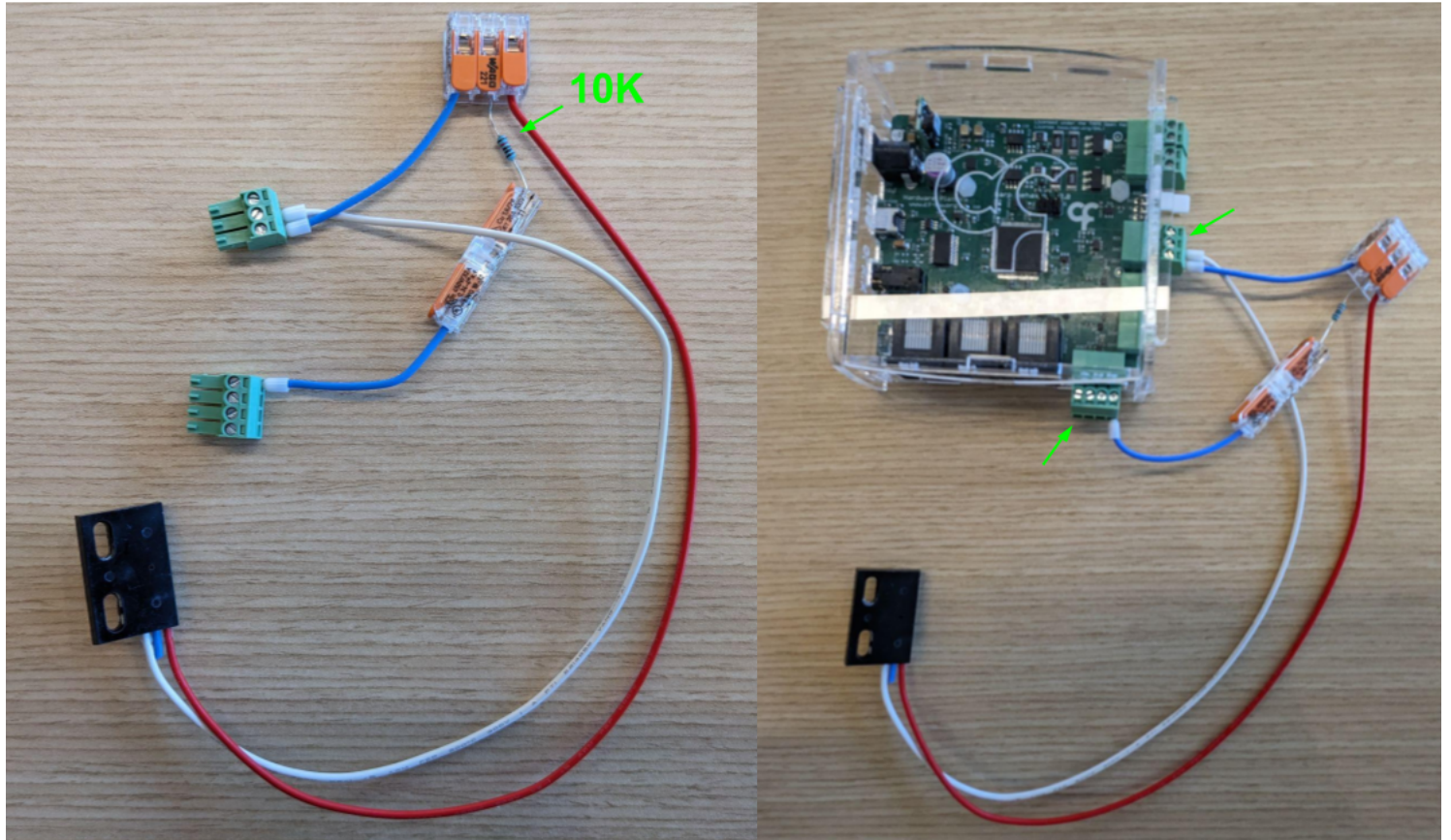
You will notice that despite the timestamp being present in the message, the **AnalogInput0** output stream is not timestamped. This is because the **Parse** operator does not propagate the timestamp from the original message by default. In cases where the timestamp is necessary, for each **<Payload>** 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)
- The output type of `Parse` will now change and propagate the state of all lines according to the demultiplexing logic of the register:

```
DigitalInputs:
  bits:
    None: 0x0
    DIPort0: 0x1
```

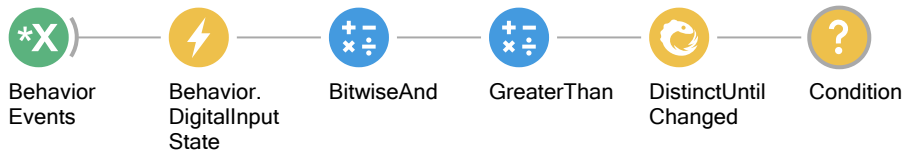

DIPort1: 0x2

DIPort2: 0x4

DI3: 0x8

In other words, each bit of the integer value of the register represents the state of a specific line. If only DI3 is currently High, we would get: $00001000 = 8 = 0x8$ Conversely, if DI3 and DIPort0 are High, we would get: $00001001 = 9 = 0x9$

- To extract the state of a specific line, use the BitwiseAnd operator and Value to the line you want to extract (e.g. DI3). To convert to a Boolean, use the GreaterThan operator with Value set to 0.
- 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.

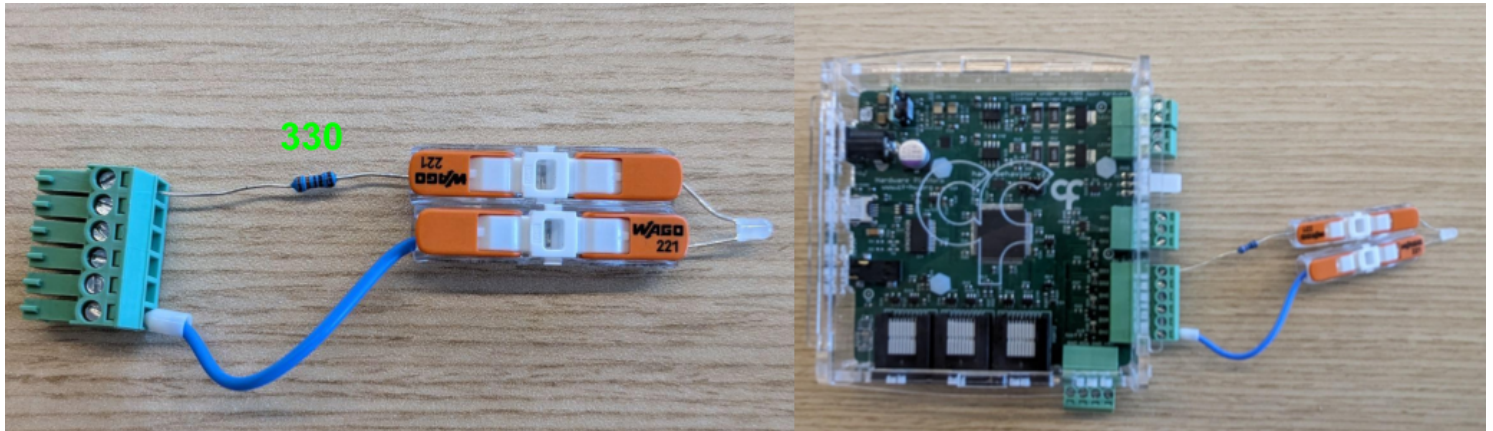


NOTE

In most situations listening to the Event propagated by the register is sufficient, and preferred, to keep track of the full state history of the device. Alternatively, one could also switch to a "pooling"-like strategy by using a Timer operator that periodically asks for a Read from the register.

Sending Commands to the device

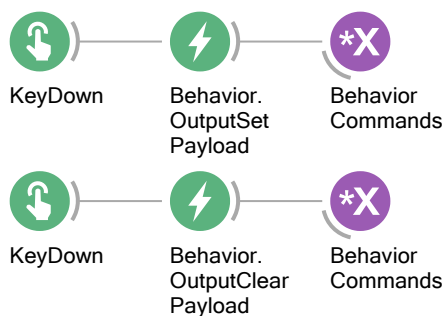
Assemble the following example:



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 `D03` to `High`:

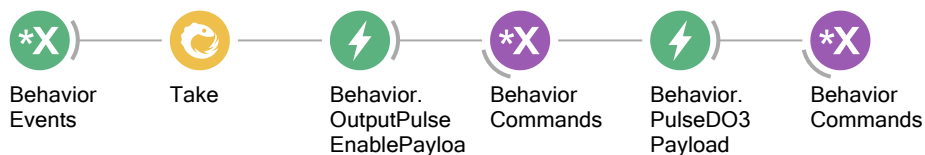
- Add a `KeyDown(Windows.Input)` operator and set the `Filter` property to a specific key (e.g. `1`).
- Add a `CreateMessage(Harp.Behavior)` operator in after the `KeyDown` operator.
- Select `OutputSetPayload` under `Payload`. Make sure the `MessageType` is set to `Write` since we will now be asking 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. `D03`).
- Replicate the previous steps to clear (turn off) the state of the line `D03` 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 `D03` 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. **D03**), 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 **D03** every time you press the key **1**.



NOTE

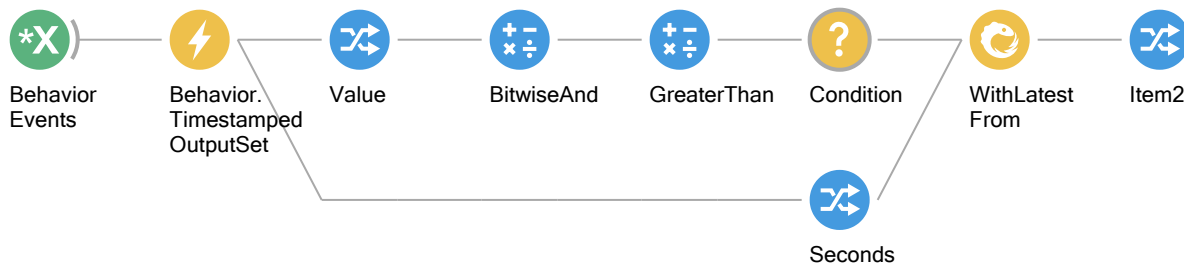
The **BehaviorEvents->Take(1)** pattern will wait for the first message from the device before sending any commands, guaranteeing that the device is ready to receive commands.

Getting the timestamp of a Write message

While we know that the state of the line **D03** 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 **BitwiseAnd(D03)** and a **GreaterThan(0)** operator, after **Value** to extract the state of the line **D03**.
- Add a **Condition** operator to only allow **True** values to pass through (since we are only interested in changes of **D03**).

- Recover the initial timestamp of the message by using a **WithLatestFrom** operator connecting the output of **Condition** and **Seconds**.



NOTE

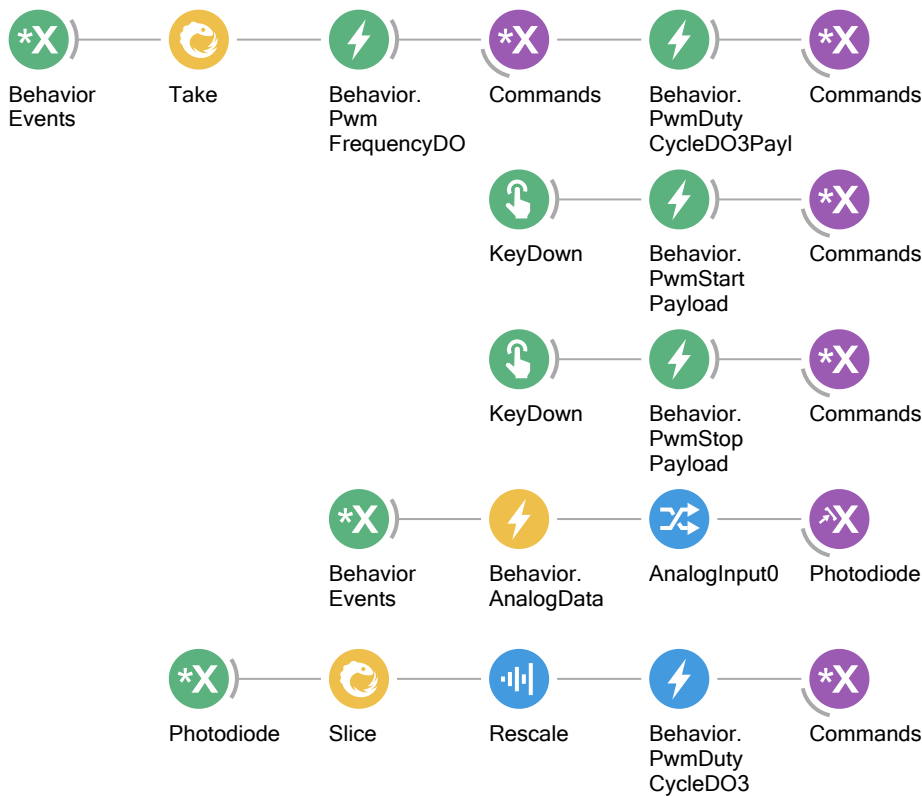
More documentation on how to manipulate timestamped messages can be found [here](#)

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 **D03** 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. **D03**).
- 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.

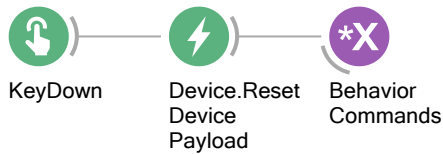
- Subscribe to the **Photodiode** stream and add a **Rescale** operator. According to the [documentation of the Harp Behavior board](#), the duty cycle register only accepts values between 1 and 99. As a result, we need to rescale the value of the ADC channel to match this range. Set the **Max** and **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.



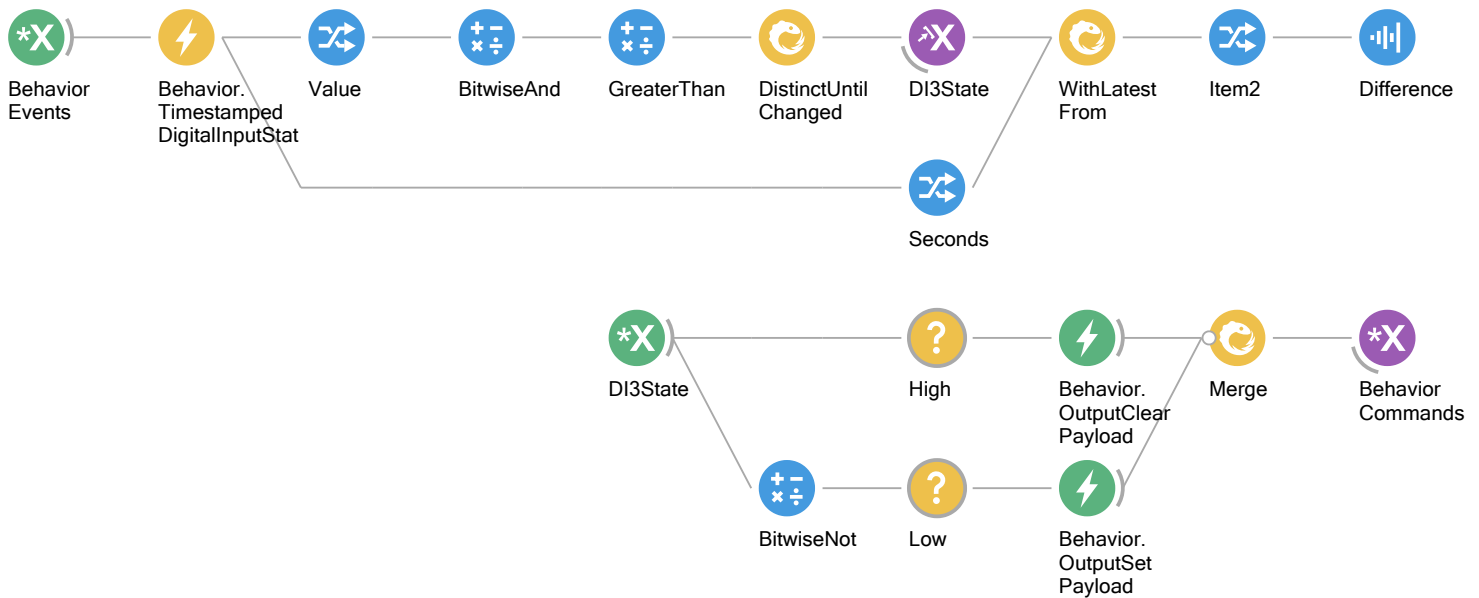
Benchmarking round-trip time

As a final example, we will show how to measure the [round-trip time](#) of a message sent to the device. This is useful to understand the latency of the closed-loop system and to ensure that the system is running as expected. The idea is to send a message to set the state of a digital output line, wait for the reply (t1) message, and invert the state of the line once this message is received, once again waiting for the second, corresponding, reply (t2). By calculating t2-t1, we will have the time it takes for a message to be sent from the device, processed by the computer and received again by the device:

- Connect **D03** to **DI3** with a jumper cable.
- Read the timestamped values from the **DI3** pin using **DigitalInputState**:
- Subscribe to the **BehaviorEvents** stream. Add a **Parse(Harp.Behavior)** operator and set the **Register** to **TimestampedDigitalInputState**. Expose the **Value** and **Seconds** members of the output structure.
- Add a **BitwiseAnd(DI3)** and a **GreaterThan(0)** operator, after **Value** to extract the state of the line **DI3**. Add a **DistinctUntilChanged** operator to only propagate the message if the state of the line of interest changes. Publish this value to a **PublishSubject** named **DI3State**.
- Recover the timestamp of the message using a **WithLatestFrom** operator connecting the output of **DistinctUntilChanged** and **Seconds**.
- Add a **Difference** operator to calculate the time between the two messages (i.e. **t2-t1**).

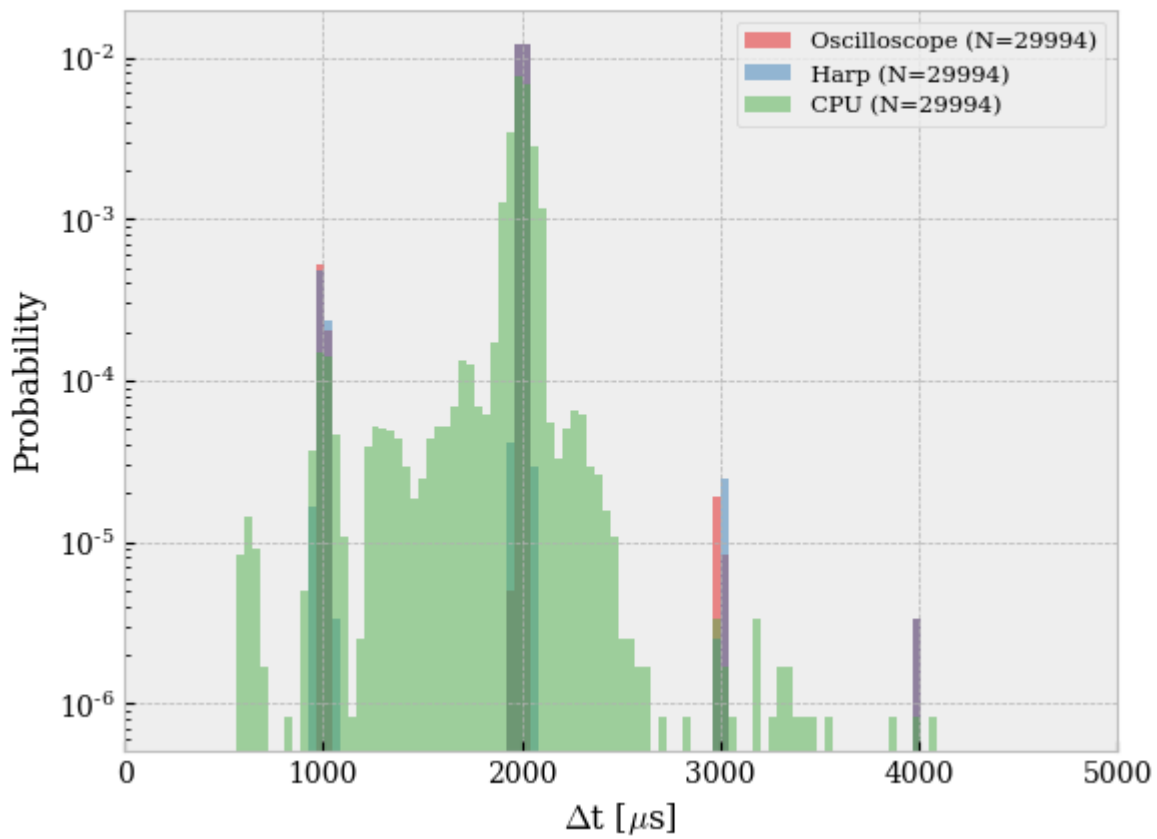
Now that we have the state of the input line, we need a way to close-loop it with the output line.

- Subscribe to the **DI3State** stream;
- Make two branches from this stream, to set-up a **if-else**-like statement.
- To the first branch, add a **Condition** that will take care of the case where the state of the input line is **High**. Add a **CreateMessage(Harp.Behavior)** operator and set it to **OutputClearPayload** to turn off the line **D03**.
- To the second branch, add a **BitwiseNot** followed by a **Condition** operator to take care of the case where the state of the input line is **Low**. Add a **CreateMessage(Harp.Behavior)** operator and set it to **OutputSetPayload** to turn on the line **D03**.
- Join the two branches with a **Merge** operator, and propagate the message to the device using a **MulticastSubject**.
- Run the workflow and check the output of the **Difference** stream.



NOTE

The timestamps reported by Harp can be independently validated by probing the digital output line and calculating the time between each toggle. We have done this exercise in the past and found that the timestamps closely match.



Source	Mean[μs]	Std[μs]	Min[μs]	Max[μs]	1%[μs]	99%[μs]
Oscilloscope	1972.1	174.1	985.0	4002.0	991.0	2019.0
Harp	1972.1	174.0	959.9	4000.2	991.8	2016.1
CPU	1971.7	171.9	576.0	4057.0	1011.0	2240.0

Logging

Logging messages from device

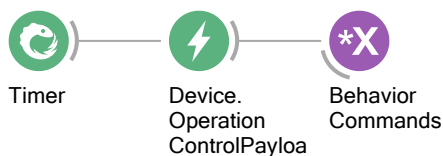
- Subscribe to **BehaviorEvents**
- Add a **GroupBy(Bonsai.Harp)** operator
- Add a **MessageWriter** operator. Set the **FileName** property to **MyDevice.harp/Behavior.bin**



Ask for a device register read-dump

It is critical that the messages logged from the device are sufficient to reconstruct its state history. For that to be true, we need to know the initial state of all registers. This can be asked via a special register in the protocol core: [OperationControl](#). This register has a single bit that, when set, will trigger the device to send a dump all the values of all its registers.

- To the previous example, in a different branch:
- Add a **Timer** operator with its **DueTime** property set to 2 seconds. This will mimic the delayed start of an experiment.
- Add a **CreateMessage(Bonsai.Harp)** operator after the **Timer**
- Select **OperationControlPayload** under **Payload**. Depending on your use case, you might want to change some of the settings, but we recommend:
 - **DumpRegisters** set to **True** (Required for the dump)
 - **Heartbeat** set to **True** (Useful to know the device is still alive)
 - **MuteReplies** set to **False**
 - **OperationLed** set to **True**
 - **OperationMode** set to **Active**
 - **VisualIndicator** set to **On**
- Add a **Multicast** operator to send the message to the device



⊗ IMPORTANT

In your experiments, always validate that your logging routine has fully initialized before requesting a reading dump from the device. Failure to do so may result in missing data.

Completing the logging pattern with the `device.yml` configuration file

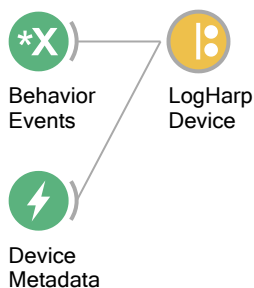
In order to use [harp-python](#) data interface to its full extent, we need to provide a `device.yml` configuration file. This file will contain the device's register map, which is necessary to interpret the data logged from the device.

This file can be manually added to the root of the logged data folder, or it can be saved in Bonsai:

- To the previous examples, in a different branch:
- Add a `DeviceMetadata(Harp.Behavior)` operator
- Add a `WriteAllText` operator to save the metadata to a file named `device.yml`



We can wrap all the previous patterns in a single grouped node:



Stay tuned for updates as, while the logging spec has been defined, the `Bonsai.Harp` library will soon be updated to include operators to more easily implement these patterns!


Data Interface

Collecting data

Use the last workflow from the previous section to collect data from your Harp Device. Additionally, if you have another Harp Behavior board and one Clock Synchronizer board, attempt to collect data from both devices simultaneously from two computers:

- Connect the **ClkOut** line from the Clock Synchronizer to the **ClkIn** line of the Harp Behavior board(s).
- Connect a button/switch to one of the digital input lines of the Harp Behavior board(s).
- Start the workflow and log all the data from the device to be analyzed later.

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
```

Analyzing single register data

A single register, in its rawest form, can be parsed as follows:

```
import harp
file = "./data/Behavior_32.bin"
data = harp.read(file)
```

Analyzing data with a device.yml file

If you have access to the **device.yml** file, you can parse the data as follows:

```
import harp
device = harp.create_reader("./data/device.yml")
file = "./data/Behavior_32.bin"
data = device.DigitalInputState.read(file)
```

Analyzing data using the recommend logging spec

If you follow the recommend logging spec covered at the end of last section, where the device.yml is in the same folder as all registers, you can parse the data as follows:

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

Verify that both behavior boards are synchronizer

- Using the `DigitalInputState` register, parse the value of the button/switch and verify if both boards are synchronized (i.e. report the same timestamp for the same button press).