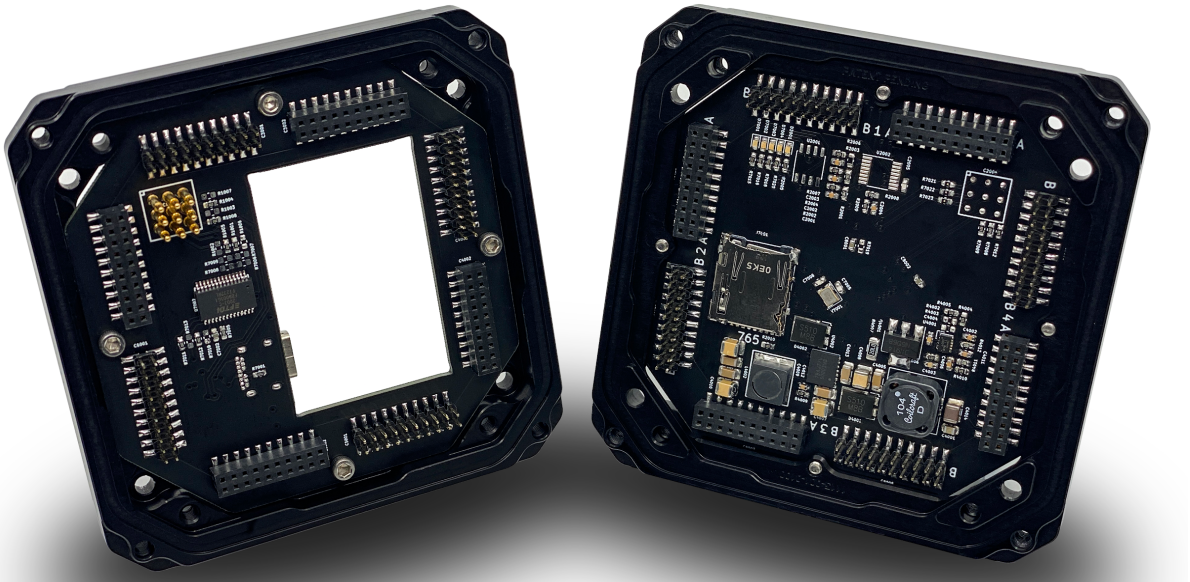
NODEUDESIGN

Brain & Programmer
Getting Started

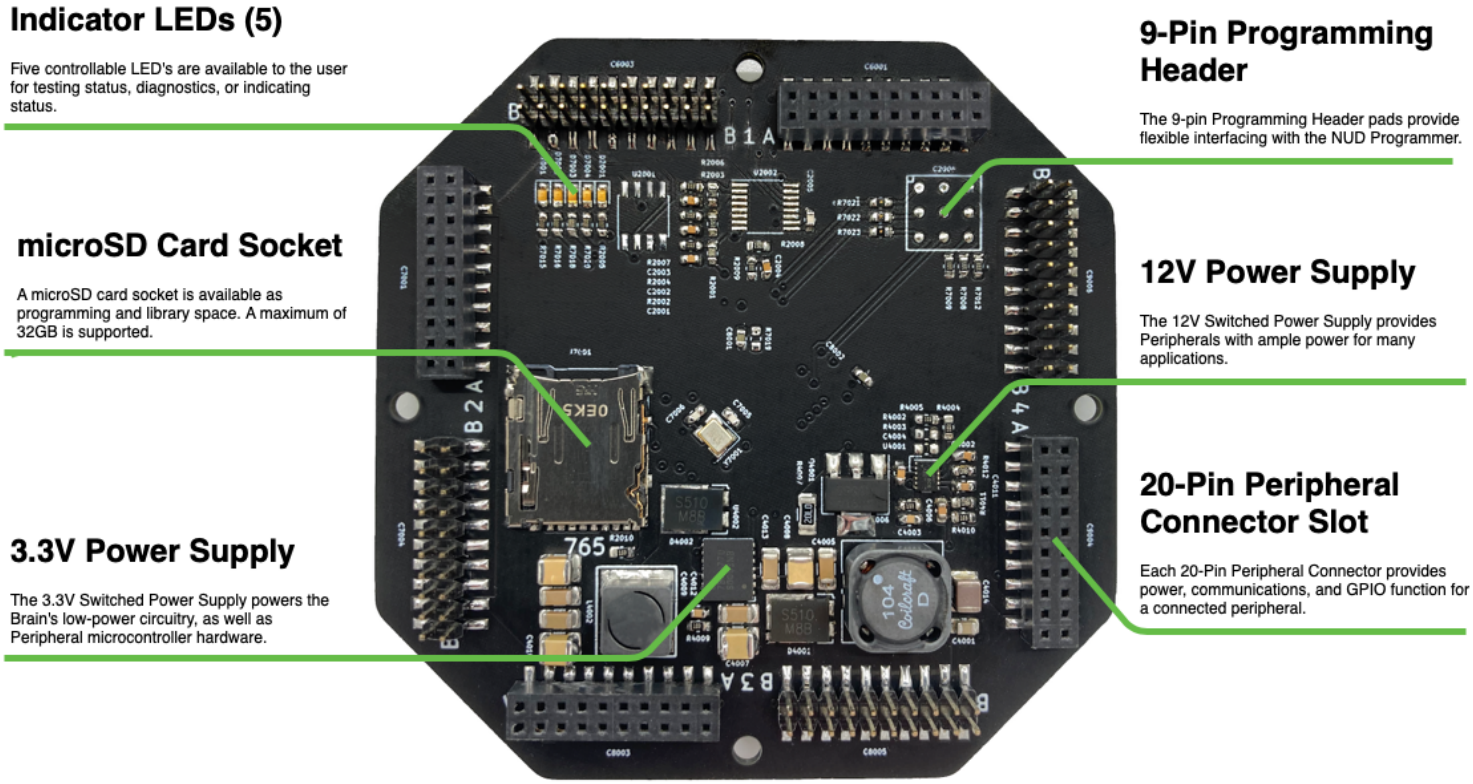


https://www.nodeudesign.com
sales@nodeudesign.com

Table of Contents

1. Introduction to the NUD Brain

1.1. Layout



Programing Header

The 9-pin programming header allows connection to a programming board. The Brain programming contacts give connection to USB, serial, programming, and reset.

Power

The Brain requires a voltage between 7.5V and 36V or a connection to a programmer connected to a USB port. The Brain contains two onboard power regulators. The first enables the logic of the Brain and any attached Peripheral, the second enables functions of the peripherals. Power for both logic and functional applications may be supplied through a connector peripheral. Additionally, Brain and Peripheral logic may be powered through the Programmers USB-C interface.

Note: Only Brain and Peripheral logic operation is supported when powering the system through a USB-C connector. Functions through a connector peripheral may NOT work properly!

uSD Card

The Brain incorporates a uSD card slot supporting up to a 32GB uSD card. The card is used to store code and contain log files.

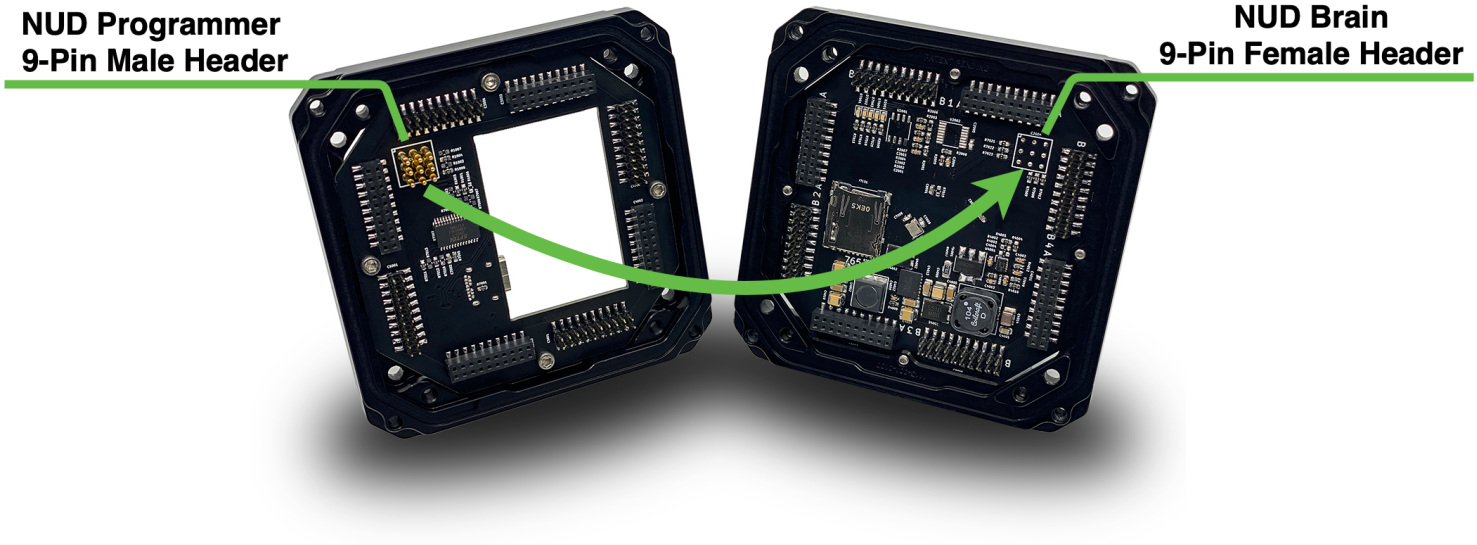
LEDs

The Brain contains 5 LED’s. LED 1, the LED nearest to the edge of the PCB, indicates that the brain is powered. LEDs 2-5 are available to the user for indicating status, diagnostics, or any other user preference.

1.2.Connecting to Computer

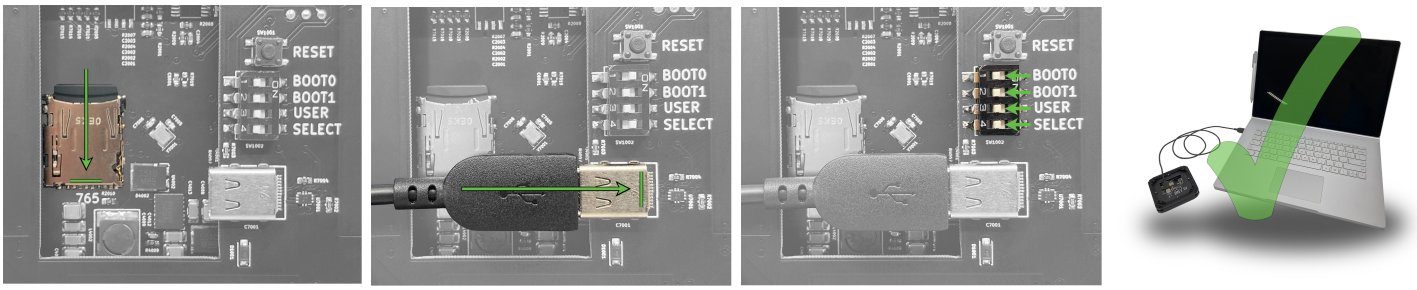
Connect a Programmer Peripheral to the Brain:

Align the Programmer's 9-pin programming header to the programmer contacts on the bottom side of the Brain. Fully press the Programmers perimeter connectors into the Brains perimeter connectors verifying all pins are aligned.



Accessing the Flash Memory on the brain:

Insert the microSD Card into the Brain microSD card slot. Connect a USB-C Cable to the Programmer USB-C socket. Locate the DIP Switch package on the Programmer and slide all switches to the position nearest to the opening of the programmer. This configuration tells the Brain to Run Boot 0, and communicate in USB Mode. The computer should then recognize the Brain as a flash drive and a virtual COM port. If the computer does not immediately recognize the Brain, try pressing the reset button. If multiple reset attempts fail, remove and replace the Programmer back in its original position, making sure to align the Programmer's programming headers with the Brain's programming contacts.



Once the brain has successfully connected to the computer, the drive Brain should mount to the computer exposing the uSD card filesystem. Within the uSD card, there should be two files: boot.py and main.py.

- boot.py is executed at startup and contains various configuration options for the Brain.
- main.py is the python script that is run immediately after boot.py. The initial file to be run after boot may be changed in boot.py.

1.3.Running First Script

Open the main.py file in your preferred text editor such as Sublime Text, Notepad++, or VIM. With the file open on a fresh Brain, there should be 1 line as shown below:

```
# main.py -- put your code here!
```

This line starts with a # character, which means that it is a comment. Such lines will be ignored at runtime, have no effect on the code being written, and are there for you to write notes about your program.

Add 2 lines to this main.py file, to make it look like this:

```
# main.py -- put your code here!
import nud
nud.LED(1).on()
```

The first line of code says that we want to use the nud module. This module contains all the functions and classes controlling the features of the Brain. The second line of code turns the first LED on. It initially gets the LED class from the nud module, creates LED number 1 (the first LED), and then turns it on.

In order to run the code that was written, save and close the main.py file. Next, eject (or unmount) the NUD USB drive by right clicking on the drive in the file explorer window, and selecting “eject”. Finally, press the Reset button on the programmer to reset the Brain. The LEDs will flash in the power up sequence and LED 1 should turn on and stay on.

Congratulations! You have written and ran your very first Python program!

1.4.REPL (Read-Eval-Print-Loop)

What is REPL?

REPL is the interactive Python interface that can be accessed on the Brain. Using REPL is the easiest way to test code and run commands. REPL can also be used in addition to writing scripts in main.py. How REPL is utilized is dependent on the users operating system.

How to Connect to REPL:

On a Windows computer, a USB driver package may need to be installed in order to interface with the REPL system.

Follow these steps to install USB Serial drivers:

1. Open your computer's Device Manager and locate the pyboard USB device in “Other Devices”. Right click on the device, and click “Update Driver Software”
2. A window should appear asking how you would like to install the driver. Select “Browse for driver software on your computer”
3. Navigate to the pyboard’s directory, and select that. A window may appear stating “Windows cannot verify the publisher of this driver software”. This is normal. Click “Install this driver software anyway”.
4. The driver should now install automatically. When the installation has finished, the window should say Windows has successfully updated your driver software.

For Windows users, once drivers are installed and the COM port is visible in the Device Manager, run a familiar terminal program and open the COM port. The correct baud rate is 115200. If the COM port is not visible, try resetting and reconnecting. See “*1.4.3 Resetting the Board*”. For most users, PuTTY is recommended as it offers users a free and standard terminal interface. In order to connect to the COM port using PuTTY, click on “Session” in the left-hand panel, then click the “Serial” radio button on the right, then enter the COM port (eg COM4) in the “Serial Line” box. Finally, click the “Open” button.

Note: when first opening the serial program (PuTTY, screen, picocom, etc) a blank screen with a flashing text cursor may be all that is visible. Press Enter and a Python prompt should be presented, i.e. >>>. A simple test shown below can be used to verify correct operation.

Using REPL

Through the serial terminal, code may be entered and run immediately on the Brain. REPL works by reading the command you give it, evaluating its meaning, processing the necessary data, and printing the output back to you! Here is an example:

```
      >>> print("Hello NUD!")
Hello NUD!
```

The above code shows how a user may enter “print(“Hello NUD!”) and the Brain will immediately respond back with “Hello NUD!”.The >>> characters indicate that the Brain is ready to receive and execute code or commands. Math operations, NUD hardware operations, and more can be run through REPL. Below are a few simple examples:

```
>>> import nud
```

```
>>> nud.LED(1).on()
>>> nud.LED(2).on()
>>> 1 + 2
3
>>> 1 / 2
0.5
>>> 5 * 'NUD '
'NUD NUD NUD NUD NUD '
```

Resetting the Board

There are many possible reasons to reset the Brain. The most common usage of reset is when the user modifies code. Any time code is modified or updated, the Brain must be reset in order to re-compile and execute the code. This can be done in one of two ways. The first reset is a soft reset performed by pressing CTRL-D at the Python REPL prompt. A message similar to the following will be printed upon a successful reset.

```
>>>
MPY: sync filesystems
MPY: soft reboot
Micro Python v1.0 on 2014-05-03; NUDv1.0 with STM32F765
Type "help()" for more information.
>>>
```

Sometimes things go wrong. Code can become stuck in an endless loop or REPL can become unresponsive. If a soft-reset doesn't remedy the problem, a hard reset (turn-it-off-and-on-again) can be performed by pressing the RST button (the small black button near the DIP switches on the board). This essentially turns the Brain off and turns it back on again. This will end any current REPL session or running program, and fully resets the Brains hardware. Any code should then re-compile and begin executing normally.

Note: Before a hard-reset is performed, it's recommended to first disconnect the serial terminal program and eject the NUD Flash drive.

REPL Features

REPL offers many features that make testing and development easier. One of the features is Auto-Indentation when creating nested statements like if-else conditionals or for loops. The following example shows how the cursor for the next line of code is automatically indented if an if statement is typed into REPL. This indentation format is required for the Python language. To get out of the indentation, simply hit backspace on an empty line and the cursor will be moved out one indentation level.

```
>>> if(n==4):
...     |
```

If a program is already running, REPL can interrupt and stop the program to allow terminal access to the Brain. This can be done by pressing CTRL-C in an open serial terminal. This will terminate any program that is running. The program can be restarted by issuing a soft-reset by pressing CTRL-D.

1.5.LED's, Timing, and Simple Python

As was demonstrated in the previous section, LEDs can be utilized in the REPL. First consider the following code entered into the REPL terminal.

```
>>> myled = nud.LED(1)
>>> myled.on()
>>> myled.off()
```

The first line creates an object and assigns it to the first LED of the Brain. The second and third lines simply turn on and off the LED.

This is all very well, but this process could be automated through scripting. Open the file main.py on the pyboard in a text editor. Type or paste the following lines into the file. Make sure the indentation is correct since this matters! Unlike many languages, Python uses indentation to denote nested code.

```
import nud
import time

myled = nud.LED(2)
while True:
    myled.toggle()
    time.sleep(1)
```

Once the code is entered, save main.py and press CTRL-D in the serial terminal to perform a soft reset. When this script runs, the second LED on the Brain should turn on for about one second and then off for about one second. To stop the script from running, and stop the LED from flashing, press CTRL-C at your terminal.

So what does this code do? First, some terminology. Python is an object-oriented language, and almost everything in python is a class. When an instance of a class is created, an object is created and stored in memory. Classes have methods associated with them. A method (also called a member function) is used to interact with or control the object.

The first line of code imports the nud module which gives access to the core hardware interface of the Brain. The second line imports the time module. The time module gives access to the internal timing of the Brain. Time includes getting the time since the Brain powered on and allows putting the Brian to "sleep" as a delay. Note: interrupts are still active when sleeping.

The third line creates an LED object which was called myled. When the object is created, it takes a single parameter which must be between 1 and 4, corresponding to the 4 LEDs on the Brain. The nud.LED class has three important member functions that can be used: on(), off() and toggle(). The other function that was used is nud.delay(). This function simply waits for a given time in milliseconds. The statement while True: creates an infinite loop which toggles the led between on and off and waits for 1 second.

- Exercise: Try changing the time between toggling the led and turning on a different LED.
- Exercise: Connect to the Brain through REPL, create a nud.LED object and turn it on using the on() method.

So far only a single LED has been used, but the Brain has 4 available (not counting the power LED). Start by creating an object for each LED to control each of them. Do this by creating a list of LEDS with a list comprehension.

```
myleds = [nud.LED(i) for i in range(1,5)]
```

If you call nud.LED() with a number that isn't 1,2,3,4 an exception will be thrown. Next set up an infinite loop that cycles through each of the LEDs turning them on and off.

```
n = 0
while True:
  n = (n + 1) % 4
  myleds [n].toggle()
  time.sleep_ms(250)
```

Here, n is a variable used to keep track of the current LED. Every time the loop is executed, the code will cycle to the next n (the % sign is a modulus operator that keeps n between 0 and 3.) Then we access the nth LED and toggle it. If you run this you should see each of the LEDs turning on then all turning off again in sequence.

One problem that may be found is that stopping the script and restarting it again will have the LEDs as they were from the previous run. This ruins the careful toggling of LEDs as was planned. This can be fixed by turning all the LEDs off when the script is initialized and then using a try/finally block. When CTRL-C is pressed, Python generates a VCPInterrupt exception. Exceptions normally mean something has gone wrong. A try: command can be used to "catch" the exception that was "thrown". In this case it is just the user interrupting the script, so there is no need to catch the error but just tell Python what to do when it exits. The finally block does this, and is used to make sure all the LEDs are off. The full code is shown below:

```
myleds = [nud.LED(i) for i in range(1,5)]
for L in myleds:
    L.off()

n = 0
try:
   while True:
      n = (n + 1) % 4
      myleds[n].toggle()
      time.sleep(50)
finally:
    for L in myleds:
        L.off()
```

2. Peripherals

Peripherals are the backbone of the NUD architecture. Peripherals control every input, output, and communication interface that can be built into a node. Peripherals are fitted into the headers along the perimeter of the Brain leading to the association of a side/slot for each peripheral. However, before a peripheral can be used, the NUD system must be created and initialized.

NUD System

The NUD system organizes the peripherals and handles the messages that are transmitted and received. Before the peripherals are created, the System should be created. The code below shows how this is done.

```
import NUD.System as sys
nudSystem = sys.System()
```

The first line imports the system module from the NUD package. The usage of “as” allows creating a simpler name “sys” for brevity. Once the system is created, by calling sys.System(), all of the NUD subsystems necessary for handling peripherals are initialized. Now, objects representing the peripherals can be created.

Creating Peripheral

For the purpose of this tutorial, the Output peripheral will be used. When creating the peripheral, there are 3 primary modules that may be imported: the Peripheral module, the Types module, and the Values module.

```
import NUD.Peripherals.Output.Output as o
import NUD.Peripherals.Output.Types as ot
import NUD.Peripherals.Output.Values as ov
```

“import NUD.Peripheral.Output.Output as o” imports the core of the Output peripheral which contains all the properties and operations that can be used with the peripheral. It also contains the events for which callback functions can be attached.
“import NUD.Peripheral.Output.Types as ot” imports the types that are used with the peripheral. It contains the complex types used when storing/fetching properties or sent as a part of an operation.
“import NUD.Peripheral.Output.Output as ov” imports the values that can be used. For example, the Output peripheral output mode can be set to a value of Digital, PWM, Voltage, or Current.

After importing the modules, the peripheral can be created as shown below. The first line simply creates the object that represents an Output peripheral. The second line attaches the Output peripheral to the Brain's system and tells the system that the peripheral is located in slot A of the first side of the Brain.

```
myOutput = o.Output()
nudSystem.attachPeripheral( myOutput, sys.SideSlot.SIDE_1_SLOT_A )
```

After the peripheral has been attached to the system, only storing properties may be performed. In order to activate the system and allow bidirectional communication, the system can be started by calling startAutoUpdate as shown below. Calling the auto update function of the system means that all communication between the Brain and peripherals can be event driven and will only be handled when there is data to handle. The automatic system update function is useful in situations where feedback is driving an event in the system, and system operation may not necessarily be sequential.

```
nudSystem.startAutoUpdate()
```

If more deterministic communication needs to be utilized, the auto update function can be left un-started. This means that the system's update function must be periodically called to completely transfer messages. This is shown in the following example which will call the update function once every second. Manual control of the system update is useful where synchronized timing is critical, or routine updating is necessary for successful operation.

```
while(1):
    time.sleep_ms(1000)
    nudSystem.update()
```

Properties

Peripherals can be configured by "fetching" and "storing" their properties.Properties can be stored at any time during the system operation. However, the results of a fetch can only be used after calling system.update(). A store action is the means by which a configuration value can be set. A fetch action is the means by which a configuration value can be retrieved. The fetch action only retrieves the value of the configuration. This takes time and as such, the value must be retrieved using the value() function. The code below shows all of these actions.

```
myOutput.properties.DesiredValue.store( 0, 250 )
myOutput.properties.VoltageValue.fetch()
voltage = myOutput.properties.VoltageValue.value()
```

Events

Events are actions driven from a peripheral that occur when certain conditions are met. For example, the output peripheral can be configured to generate an event when the voltage of an output goes above or below a certain threshold. In order to handle an event, a *callback* must be attached to the specific event. A callback is simply a function that is called at the time an event occurs. An example of writing a callback function and attaching it to an Output peripheral is shown below.

```
import NUD.System as sys

import NUD.Peripherals.Output.Output as o
import NUD.Peripherals.Output.Types as ot
import NUD.Peripherals.Output.Values as ov
```

```
# --- NUD System ---
nudSystem = sys.System()

# --- Output---
def OverVoltageTrigger_CB( periph, chan, data ):
       print("OverTrig", chan, ",", data)

myOutput = o.Output()
self.nudSystem.attachPeripheral( myOutput, sys.SideSlot.SIDE_1_SLOT_B )
myOutput.events.OverVoltageTrigger.setCallback( OverVoltageTrigger_CB )
```

By passing the method OverVoltageTrigger into setCallback(), “def OverVoltageTrigger_CB(periph, chan, data)” is called at the moment the condition within OverVoltageTrigger is met, resulting in the Brain printing “OverTrig” over REPL.

Operations

Operations are actions that the Brain requests the peripheral to perform. The Brain will handle all of the necessary operations to instruct the peripheral, and the peripheral will handle all the tasks that are associated with performing the operation. The Output peripheral does not currently have any operations that it performs, however, the Stepper peripheral does. The Stepper peripheral can be requested to perform a Homing operation which will drive the stepper motor until the “Home” switch is pressed. Calling the operation is shown in the example below.

```
myStepper = s.Stepper()
self.nudSystem.attachPeripheral( myStepper, sys.SideSlot.SIDE_1_SLOT_B )
myStepper.operations.operations()
```