Parlay Documentation

Release 1.3.25

Promenade Software, Inc

May 02, 2018

Contents

Parlay is a software framework that makes it easier to develop, test, and connect embedded devices.

With Parlay, you can control your hardware quickly and easily. You write code that maps your hardware functionality to Parlay **Items**, with **Commands** and **Properties**.

Then you can:

- Explore, poke, and prod items from our built-in brower-based User Interface
- Plot and log streamed data from sensors
- Create repeatable, automatic tests with simple-syntax Python scripts
- Easily integrate with user-facing applications, mobile apps, and more…

The core of Parlay is written in Python, so it is **cross-platform**. It can run on your Windows, OSX, or Linux PC.

Parlay can also run on your device, assuming your device runs embedded Linux or Windows.

Promenade Software, the company that created Parlay, specializes in developing software for safety-critical industries, such as medical devices, aerospace, and automotive. Parlay was built to streamline development and testing of embedded software.

CHAPTER 1

What is Parlay?

Parlay is a development framework that provides:

- a built-in browser-based user interface to poke and prod connected items
- a drag and drop UI builder that lets make a custom screen of buttons, sliders, graphs and more.
- pre-built components to connect to devices over serial, TCP/IP, ModBus, GPIB and more
- dead-simple scripting framework to write testing and development scripts in Python

Parlay serves a web application that provides a poke & prod user interface to all connected items. Via the user interface, users can send commands to and view responses from items, change item properties, and graph item datastreams. This user interface can be accessible over a network for development and testing, or served securely in production for remote diagnostics and service.

At the heart of Parlay is the "broker", which provides publish/subscribe message routing between items. Once items are connected to the broker, they are easily accessible from the user interface, scripts, and any other item.

CHAPTER 2

Documentation Contents

2.1 Parlay Documentation

Parlay is a software framework that makes it easier to develop, test, and connect embedded devices.

With Parlay, you can control your hardware quickly and easily. You write code that maps your hardware functionality to Parlay **Items**, with **Commands** and **Properties**.

Then you can:

- Explore, poke, and prod items from our built-in brower-based User Interface
- Plot and log streamed data from sensors
- Create repeatable, automatic tests with simple-syntax Python scripts
- Easily integrate with user-facing applications, mobile apps, and more…

The core of Parlay is written in Python, so it is **cross-platform**. It can run on your Windows, OSX, or Linux PC.

Parlay can also run on your device, assuming your device runs embedded Linux or Windows.

2.2 Example: Control an LED

An Item is a software object that represents a single piece of functionality. For example, an item can represent an LED on a board. An item could also represent an array of LEDs, or a temperature controller, or almost anything.

Items can communicate with each other, and it is common to build a hierarchy of items, starting with low-level items that directly control hardware, then higher-level items that control an entire subsystem consisting of several lower-level items.

Let's go through an example.

For demonstration purposes, we will ignore some complexities that we would have to address in a realistic system, so that we can understand what Parlay provides. For instance, the following code is *not thread safe*. It is not difficult to make this code usable for production, and we will do just that in a later tutorial.

2.2.1 Introduction

If you have an LED on your board that you want to control, you can create an LED item.

What things might you want to do with an LED?

- read its current state
- turn it on or off
- blink at a certain rate (maybe...)
- whatever else you think is useful...

Let's represent the on/off state of the LED with a property called "on". You can read that property to find out the current state. You can even stream that property to get automatic updates as it changes (more on that later).

Note: Parlay does not provide code that interfaces with your hardware directly. Parlay is a framework that takes your code that interfaces with the hardware, and makes it accessible and easy to control.

The code following example assumes you are running Parlay on a Raspberry Pi, using the python RPi.GPIO package.

2.2.2 Define the LED Class with a Property

```
# led.py

import RPi.GPIO as GPIO

from parlay import local_item, ParlayCommandItem, ParlayProperty, parlay_command

@local_item()
class LEDItem(ParlayCommandItem):

    on_state = ParlayProperty(default=False, val_type=bool, read_only=True)

    def __init__(item_id, item_name, channel):
        self.channel = channel

        # code to set up your specific hardware goes here
        GPIO.setup(channel, GPIO.OUT)

        ParlayCommandItem.__init__(self, item_id, item_name)
```

2.2.3 Add Basic Commands

Let's add two commands to turn the LED on and off. These will also set the *on_state* property.

```
@parlay_command()
def turn_on(self):
    self.on_state = True
    GPIO.output(self.channel, True)  # code specific to your hardware goes here

@parlay_command(self)
def turn_off():
```

(continues on next page)

(continued from previous page)

```
    self.on_state = False
    GPIO.output(self.channel, False)
```

2.2.4 Add a More Complex Command

Now that we've made it turn it on and off, let's make it do something more interesting. Let's make it blink.

We create a command for the LED item called "blink". This command should have two parameters: frequency, and the number of times to blink.

```
@parlay_command()
def blink(self, frequency_hz, num_blinks):

    # python has no type information, so all arguments are strings by default
    frequency_hz = float(frequency_hz)
    num_blinks = int(num_blinks)

    if frequency_hz <= 0 or num_blinks <= 0:
        raise ValueError("frequency_hz and num_blinks must be greater than zero")

    sleep_time = 0.5 / frequency_hz

    for _ in xrange(num_blinks * 2):
        self.on_state = not self.on_state
        GPIO.output(self.channel, self.on_state)
        self.sleep(sleep_time)
```

2.2.5 Putting it all together

Let's put it all together and show our LED Item class.

```
# led.py

import RPi.GPIO as GPIO

from parlay import local_item, ParlayCommandItem, ParlayProperty, parlay_command

@local_item()
class LEDItem(ParlayCommandItem):

    on_state = ParlayProperty(default=False, val_type=bool, read_only=True)

    def __init__(self, item_id, item_name, channel):
        self.channel = channel
        GPIO.setup(channel, GPIO.OUT)
        ParlayCommandItem.__init__(self, item_id, item_name)

    @parlay_command()
    def turn_on(self):
        self.on_state = True
        GPIO.output(self.channel, True)

    @parlay_command()
```

(continues on next page)

(continued from previous page)

```
    def turn_off(self):
        self.on_state = False
        GPIO.output(self.channel, False)

    @parlay_command()
    def blink(self, frequency_hz, num_blinks):
        frequency_hz = float(frequency_hz)
        num_blinks = int(num_blinks)

        if frequency_hz <= 0 or num_blinks <= 0:
            raise ValueError("frequency_hz and num_blinks must be greater than zero")

        sleep_time = 0.5 / frequency_hz

        # this is not thread-safe
        for _ in xrange(num_blinks * 2):
            self.on_state = not self.on_state
            GPIO.output(self.channel, self.on_state)
            self.sleep(sleep_time)
```

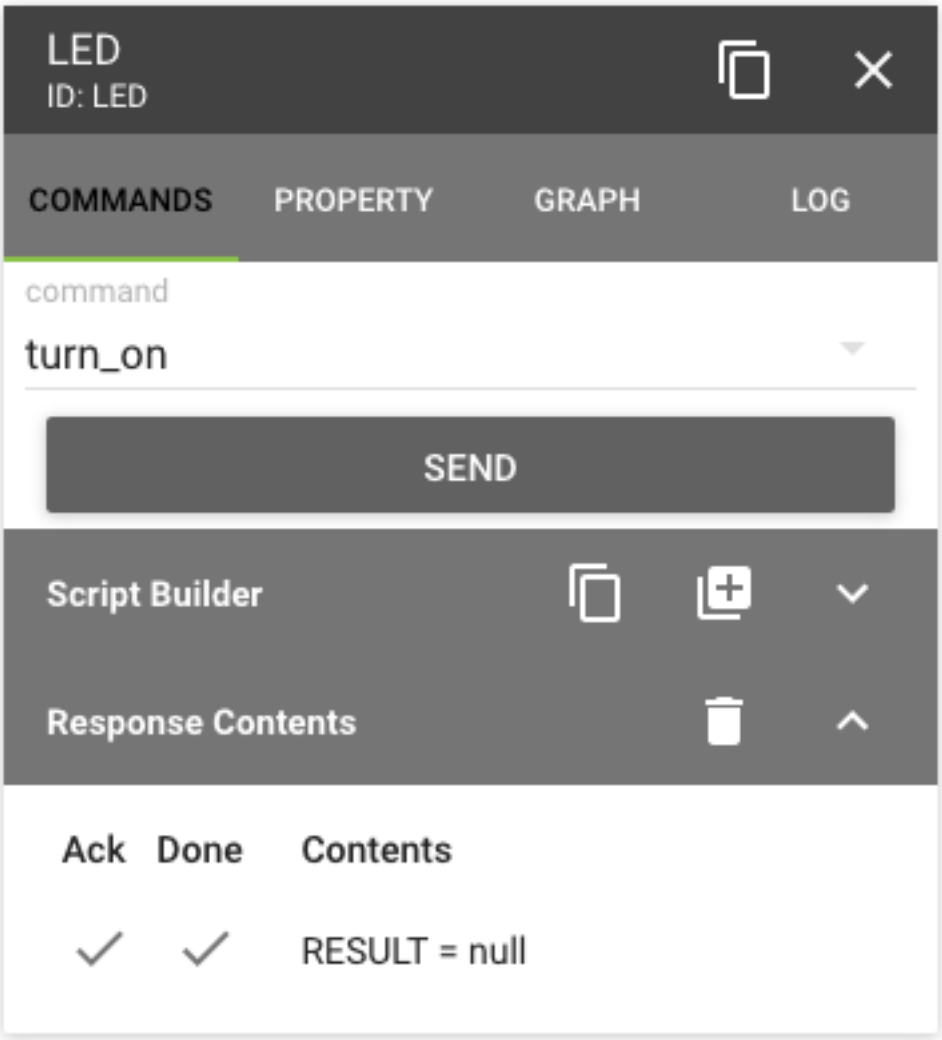
2.2.6 Instantiate our Item and Start Parlay

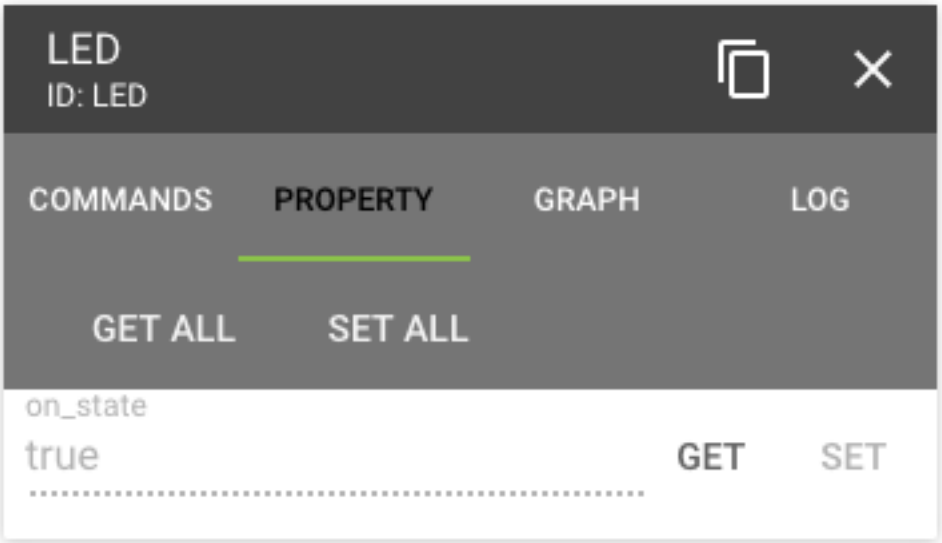
Now, let's instantiate our class and start Parlay.

```
# start_parlay_with_led.py

import parlay
import led

led.LEDItem("LED", "LED", 5)
parlay.start()
```

2.2.7 Explore the Item with the Parlay UI

Parlay includes a built-in browser-based User Interface where you can explore the commands and properties provided by your item.


LED
ID: LED
COMMANDS
PROPERTY
GRAPH
LOG
command
turn_on
SEND
Script Builder
Response Contents
Ack
Done
Contents
RESULT = null


LED
ID: LED
COMMANDS
PROPERTY
GRAPH
LOG
GET ALL
SET ALL
on_state
true
GET
SET

2.2.8 Test the Item with a Parlay Script

When Parlay is running, we can also control our item with a simple python script. This makes testing quick and repeatable. You can checkout hardware, test control sequences, and rapidly prototype functionality that you can later encapsulate in Parlay items.

```
from parlay.scripts import setup, discover, get_item_by_id, sleep

setup()
discover()

led = get_item_by_id("LED")

led.turn_on()
sleep(1)

led.turn_off()
sleep(1)

led.blink(2, 10)
```

2.3 Getting Started

Let's install Parlay and try some simple examples!

2.3.1 Install Parlay

Parlay is written in the Python programming language, for full cross-platform compatibility. As such, Parlay requires Python 2.7 to be installed your computer, as well as several python libraries.

Windows Only

Parlay requires Python and several other dependencies to be installed first.

There is a Windows installer on our GitHub page to make the process seamless.

Visit https://github.com/PromenadeSoftware/Parlay/releases/latest and click on "Parlay_<version>_Installer.exe" to download.

How to Install Parlay

You can use python's package manager, pip, to install Parlay directly from the git repository.

```
c:\> pip install parlay
```

This should install all dependencies for Parlay as well.

2.3.2 Hello World

This is the simplest demonstration of the parlay system. In this tutorial, we will create a Parlay *local_item* object, use the Parlay User Interface to send it a command, and view the response. The Item will also demonstrate the use of Properties.

This code is available as part of the Parlay Examples Repo on Github.

Create a `local_item` that says "Hello World!"

Open a text editor and create a python script file named "parlay_hello.py" like so:

```
# parlay_hello.py

import parlay

@parlay.local_item()
class CheerfulPerson(parlay.ParlayCommandItem):

    hello_message = parlay.ParlayProperty(default="Hello World!", val_type=str)
    first_name = parlay.ParlayProperty(default="Sarah", val_type=str)

    @parlay.parlay_command()
    def say_hello(self):
        return self.hello_message

    @parlay.parlay_command()
    def what_is_your_name(self):
        # properties can be used just like any variable of their value type
        return "My name is " + self.first_name + "!"


if __name__ == "__main__":

    # parlay.start() will not exit, so we need to instantiate all local items before␣
↪calling it
    CheerfulPerson("CheerfulPerson", "CheerfulPerson")
    parlay.start()
```

Save this script in any directory on your system.

Run the Hello World script

Use python to execute this script from the command line. For example, if you saved the file to the C drive, you would open a command line, and run the following command:

```
c:\> python parlay_hello.py
```

Two things will now happen:

1. The parlay system will start running in the command line. Do not close the command line window!
2. Your default web browser will automatically open a new tab and go to http://localhost:8080, where you will see the UI.

In the browser UI, you'll be able to see the "CheerfulPerson" item in the item dropdown box. Open the item and try sending the "say_hello" command. You'll see "Hello World!" as the result.

That's it! You've just created a Parlay Command Item, defined a command, run Parlay, viewed the item's card in the UI, sent the item a command, and viewed the response.

Parlay UI is a web application

This illustrates an important aspect of Parlay that can be confusing to first-time users:

Parlay runs a **web server**, *and the built-in User Interface is a* **web application**.

Parlay and the web browser are two separate applications, that in this simple example, are both running on your computer. This architecture confers huge advantages in more advanced use cases, where you can run Parlay on your embedded device. However, it is a little different than the typical stand-alone desktop application that many users are accustomed to.

- If you really want to be done with Parlay, you must close the command line window AND the browser tab.
- If you just close the web browser, Parlay is *still running*. To shut down Parlay, close the command line window where you started Parlay. If you did not mean to close the browser, you can re-open your web browser and navigate to http://localhost:8080, and Parlay will show the User Interface again.
- If you close the command line window before the web browser, the browser will lose communication with Parlay and the User Interface will show a yellow warning message "Lost Connection with Broker". Run the python script again from a command line, click "Reconnect", and the UI will be ready again.

2.4 User Interface

The Parlay User Interface (UI) is a simple and powerful tool used to visualize and interact with your Parlay Items. The UI is comprised of a workspace where you can add Item or widget Cards, and organize them by dragging and dropping each card.

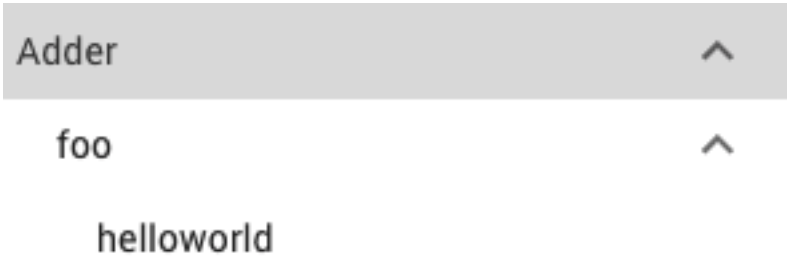
2.4.1 Items

The items scripted in Python or connected to Parlay via a custom protocol can be added to the workspace. By default, a Parlay discovery request is made to the broker system (Parlay Connect). After the discovery completes, all of your items will be made available to add into the workspace.

Adding Items into the Workspace

Clicking the items menu option in the left side navigator will pull open a library of available items. Items will only appear after a successful discovery. At the top most part of the library, there is a search bar where you can filter your Items by name or ID. Recall from the hello_world tutorial that clicking any of the items in the library will launch an item card into the workspace.

If the item is hierarchical, you will be able to expand/collapse the item and view/hide its children items by clicking the chevron icon on the right hand side of the item's displayed name/id.



After you have launched an item into the workspace, multiple functions of the item can be used. If you notice, the top of the card has the both the name and ID of the item displayed for you. On the right of the name text there will be two icons. The first icon, when clicked, will duplicated the card and display an exact copy of the item card tiled over the original. The second icon, intuitively, closes the card and removes it from the workspace. Lastly, the entire title bar of the card is used as a handle to allow you to drag and drop the card anywhere within the workspace container.

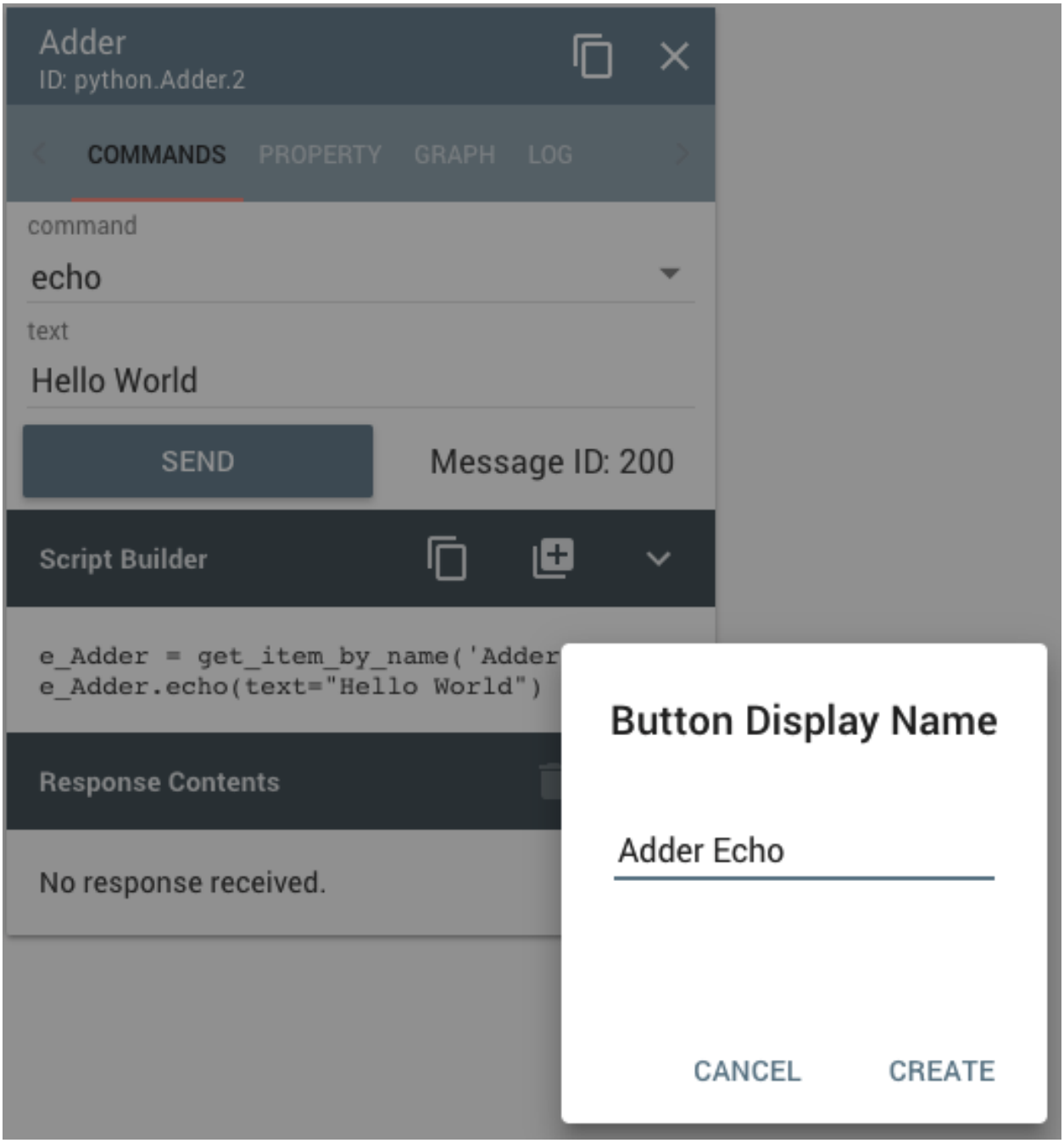
On each item card you will see a few tabs:

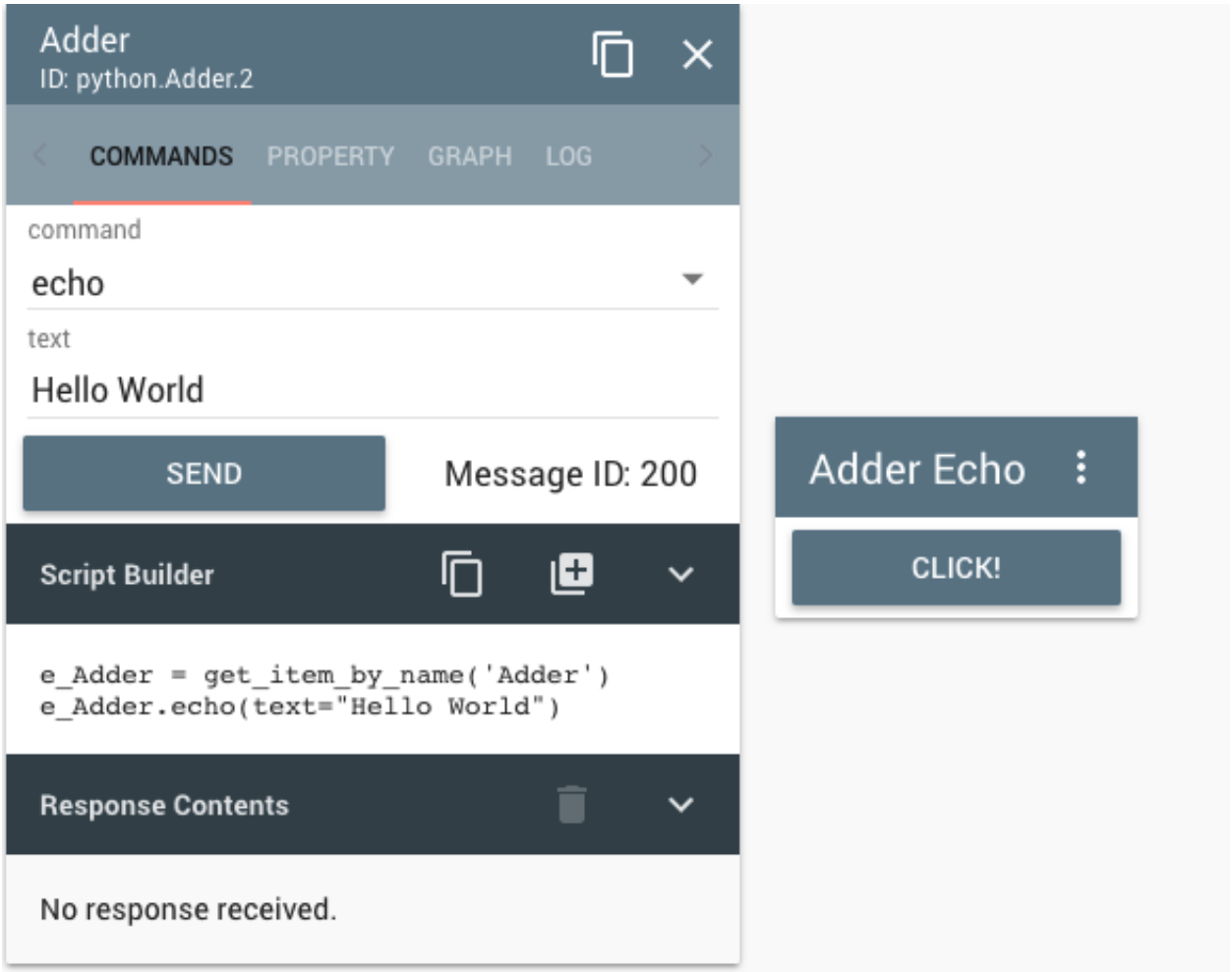
- Commands
- Property
- Graph
- Log

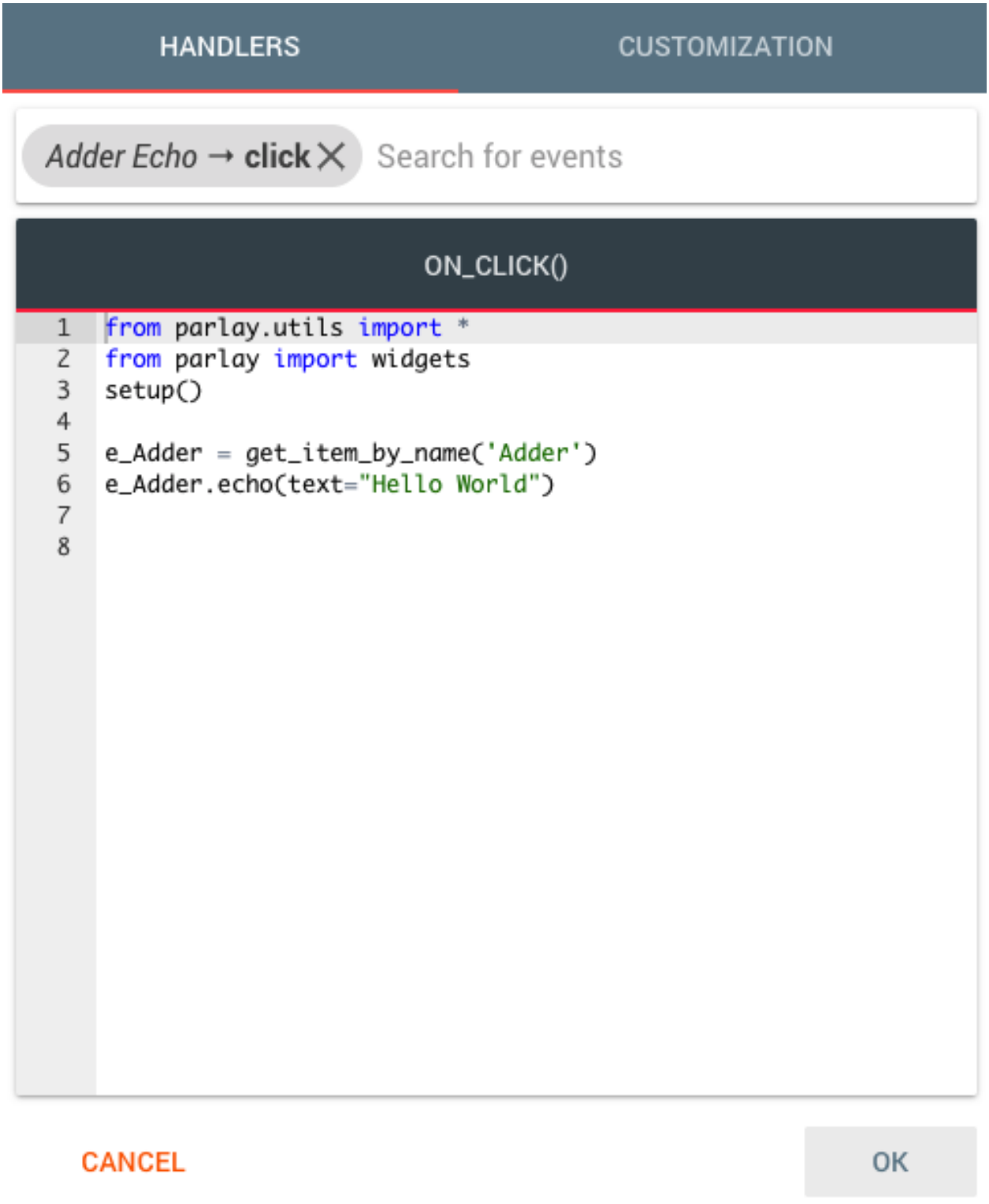
Commands

The command tab allows you to send commands/query actions to the Parlay Item. You can select different commands from the command drop down menu underneath the tab selectors. If needed, an input field will be provided to the selected command so that arguments can be passed to the command. If you recall from the hello_world tutorial, clicking the send button underneath the selected command will send the command to the Parlay system and return the response in the Response Contents section of the command tab.

Whenever the command selected changes, the contents of the script builder will also change. On the Script Builder toolbar you will see three buttons. The most useful button in this section is the middle button, which will create a button widget and copy the script and attach it to the button's click event handler.


Adder
ID: python.Adder.2
COMMANDS
PROPERTY
GRAPH
LOG
command
echo
text
Hello World
SEND
Message ID: 200
Script Builder
e_Adder = get_item_by_name('Adder
e_Adder.echo(text="Hello World")
Response Contents
No response received.
Button Display Name
Adder Echo
CANCEL
CREATE

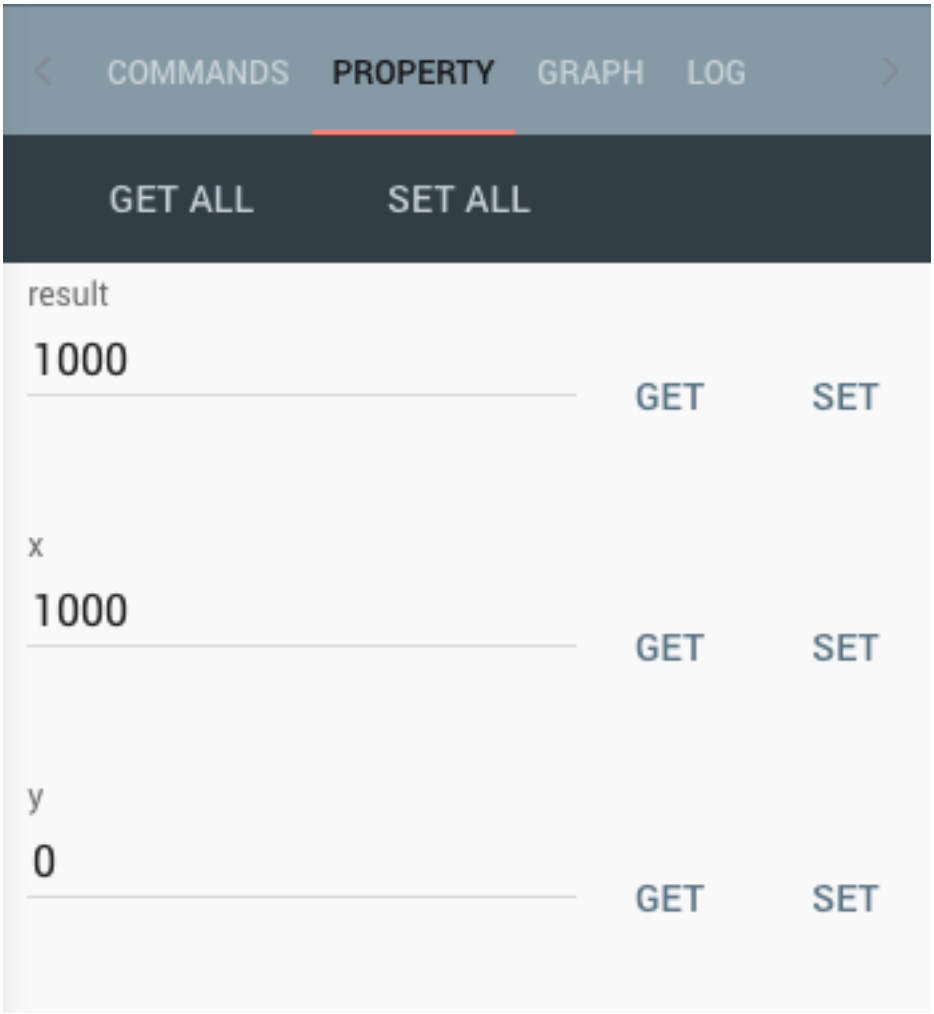

Adder
ID: python.Adder.2
COMMANDS
PROPERTY
GRAPH
LOG
command
echo
text
Hello World
SEND
Message ID: 200
Script Builder
e_Adder = get_item_by_name('Adder')
e_Adder.echo(text="Hello World")
Response Contents
No response received.
Adder Echo
CLICK!



Properties

Each item has the capability to store and hold properties, which reflect and represent different values and states of the item that you are working with. With the property tab, you can get and set properties dynamically, either individually

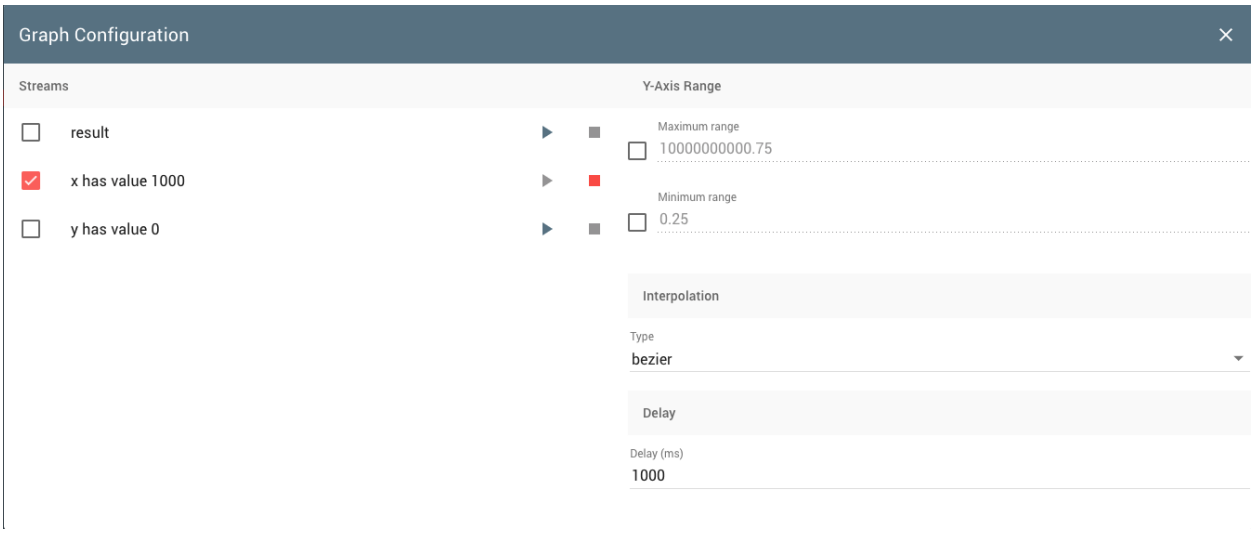
or collectively.



Note that some properties will be read only and others write only. Read only properties will have their SET button disabled and unusable and similarly, Write only properties will have their GET button disabled.

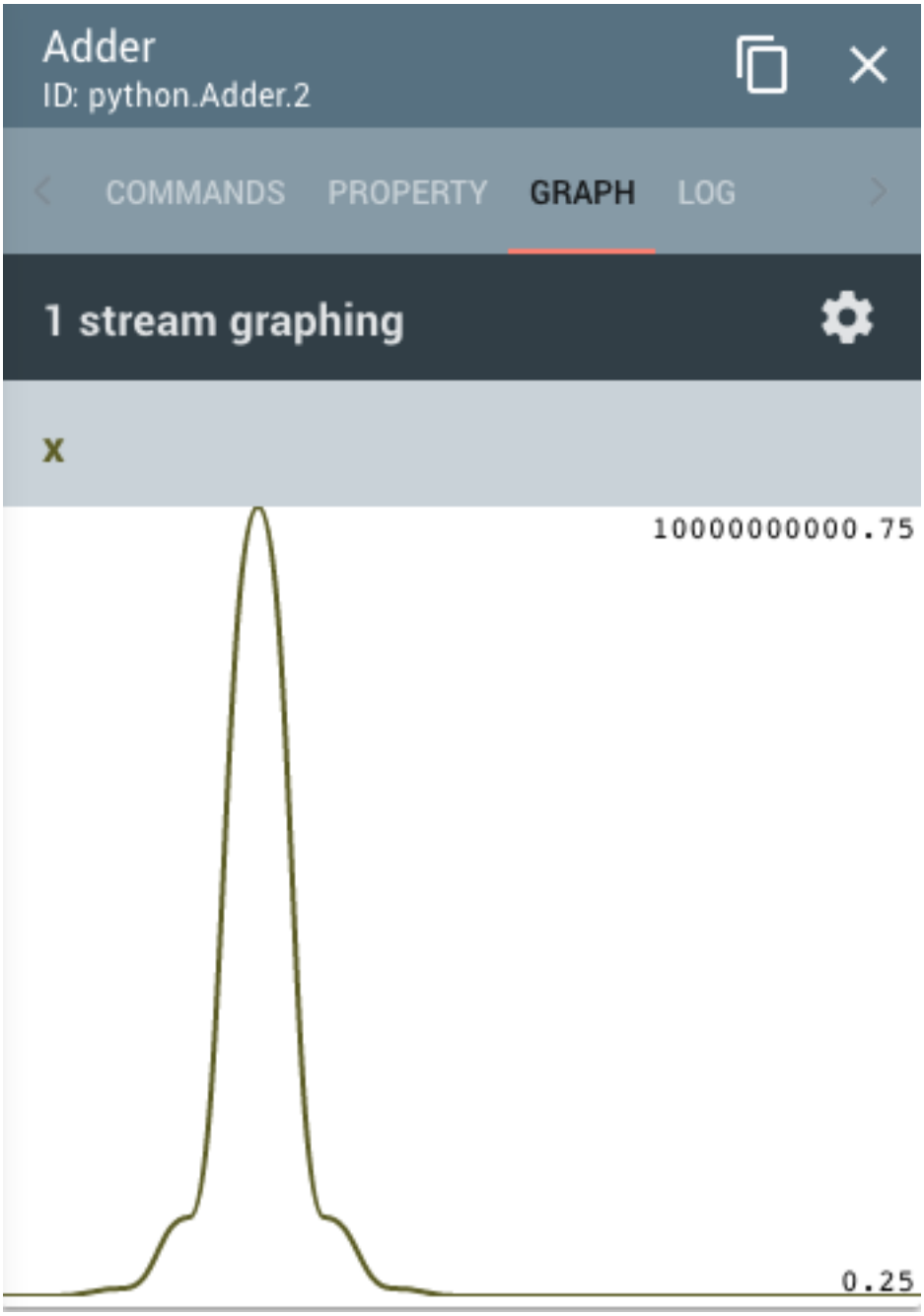
Graphs

All item properties have the capability to be streamed. This stream can be collected by the UI and streamed directly to your item card in the form of a graph. Clicking the graph tab will display a setting's cog icon. Clicking this icon will pop open a Configuration dialog for your card.



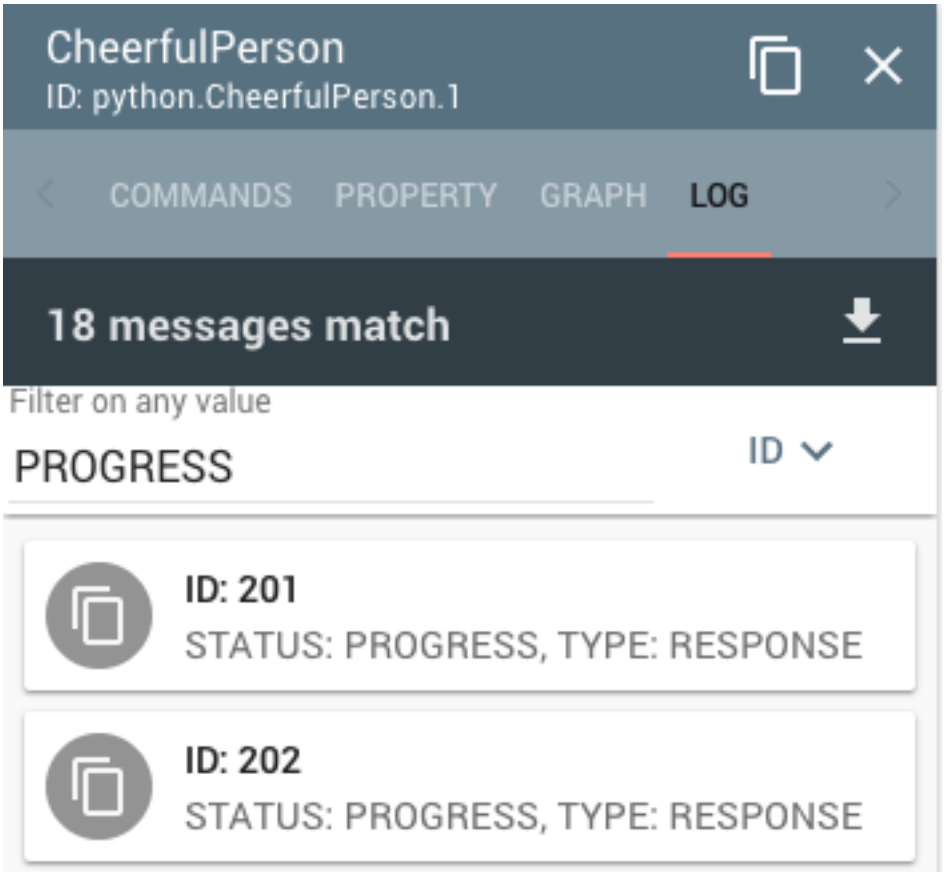
In this configuration screen, you can select which property streams will be listened to and displayed on the graph. Options to configure in this screen include the graph type (linear, bezier, step), the range of the y axis of the graph and the interval at which the item requests data.

Once the configuration is complete, you will see a graph being displayed on the card.



Logs

When commands are sent, or properties are streamed, data is being sent back and forth between the UI and the Parlay Connect (Broker) system. These data transactions occur in small JSON messages. You will see these messages in the Log tab of the item card. Through the input field on this tab, you can filter the logs based on a search string. In the picture below, we filter for "Progress" and see all the messages that contain the word progress in them.



Right next to the text that says "18 messages match" is a download icon. Clicking this icon will prompt you to download the filtered logs in a .txt file. If a filter string is not provided, the entirety of the item's logs will be saved

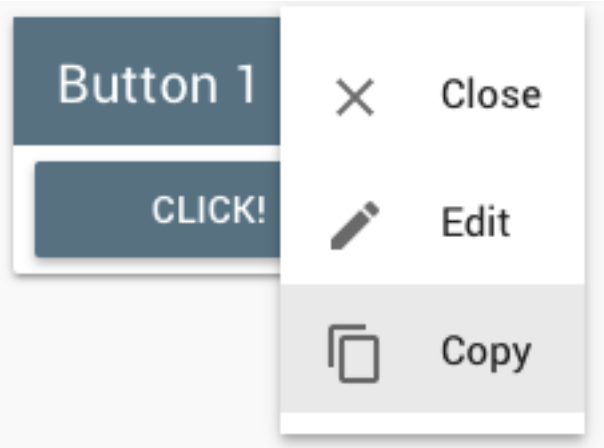
2.4.2 Widgets

Items are not the only thing that can be added to your workspace. You can also add widgets, powerful tools that contain scripting capabilities and can display data from other widgets or items.

Before diving into the functional behavior of widgets, let's take a look at how widgets are formatted. Similar to the items button on the side navigator, there is also a widgets button underneath the items button like so:

Clicking this will show all available widgets and allow you to add them to the workspace, or filter the widgets by their name. Let's start by adding a button. After clicking the button entry in the widget library we should see a widget with the name "Button <number>" in the workspace.

At the top of the widget we see a title bar. Like items, the title bar can be used to drag and drop widgets. In the upper right corner of the widget, there is an icon with three dots. Clicking this will give you a menu of actions to perform on the clicked widget.



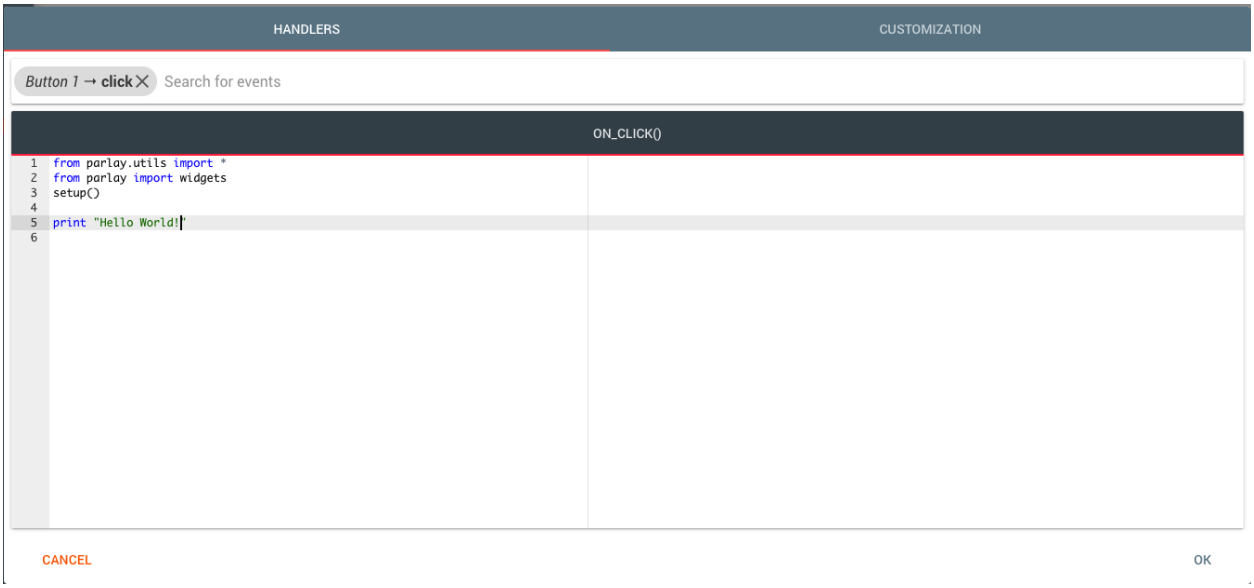
The close and copy actions are identical to that of the item card. The most important aspect of the widget actions is the edit action. Clicking the edit action will display a configuration dialog for your widget, where you can customize the widget to your liking.

Editing and Customizing Widgets

The widget edit dialog will have at most 3 tabs of the following categories: Handlers, Transformer, Customization and Script Tutorial.

Widget Event Handlers

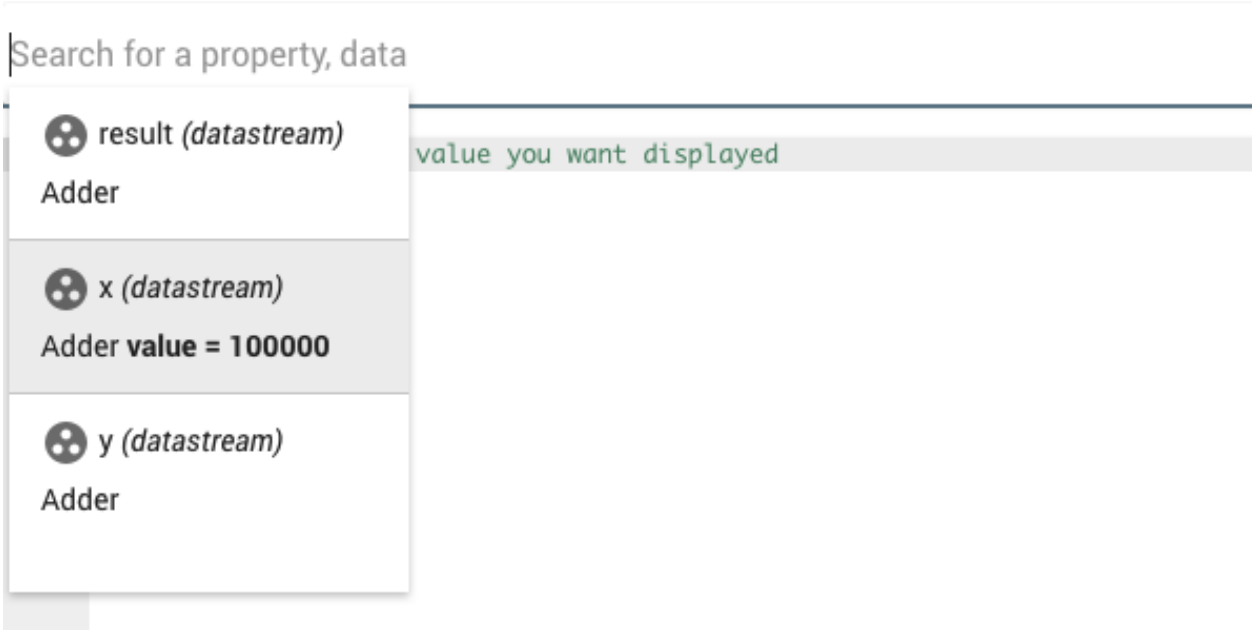
The Handler tab is only displayed for input widgets, ie, widgets that you can input actions to (button, checklist, text input, etc.). In the handler tab, you can select events that are bound to the widget and have a Python script run, every time the widget event occurs. The most common example of this is a button's "click" event. Whenever the button widget gets clicked, the Python script in the event handler will be run.



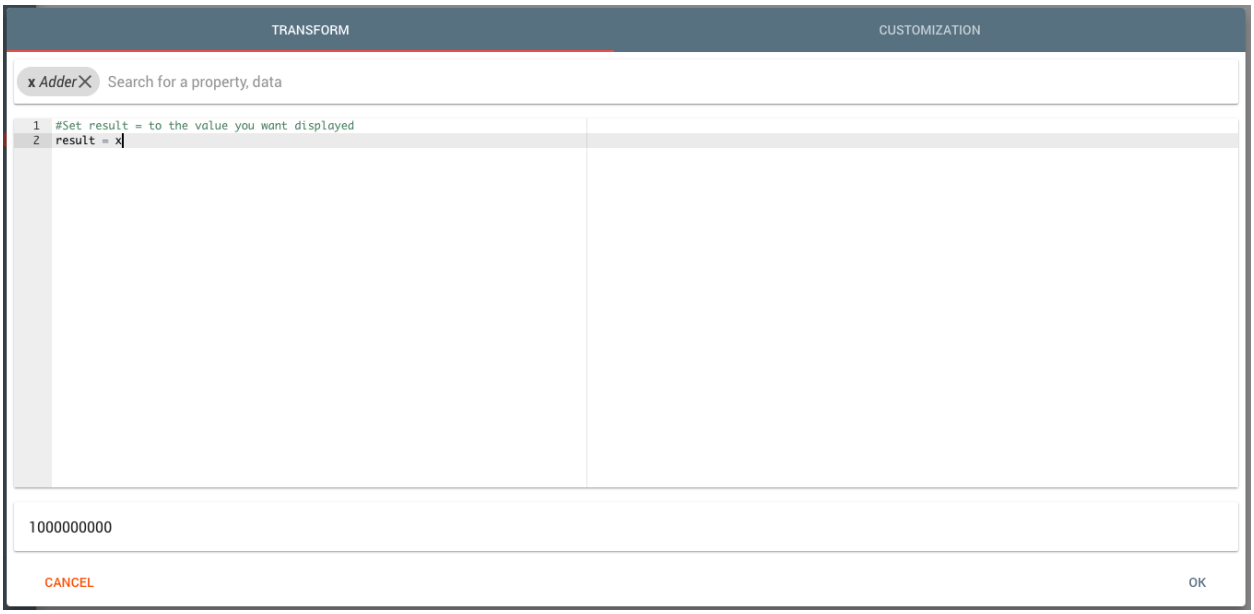
The first three lines that are appended into the script for you should never be deleted! They are needed to interact with the other Parlay Items and widgets in your workspace!

Widget Transformer

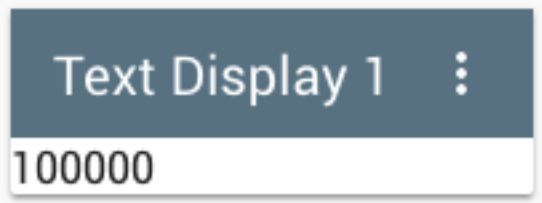
The Transformer tab is only displayed for display widgets, ie, widgets that display information to the user. Currently there is only one display type widget, the text display widget. The transformer tab lets you grab values from the Parlay scope, such as Item properties and values, and modify them as needed.



Once the desired value is formulated in the script, it can be assigned to the "result" variable and will be displayed to the screen.



The final result will be reflected in your display widget.



Widget Customization

The Customization tab is essentially how you modify the widget to your standard. Widget display names can be changed, Dimensions can be changed (such as rows/tables of a table, or size of an image button) and much more.

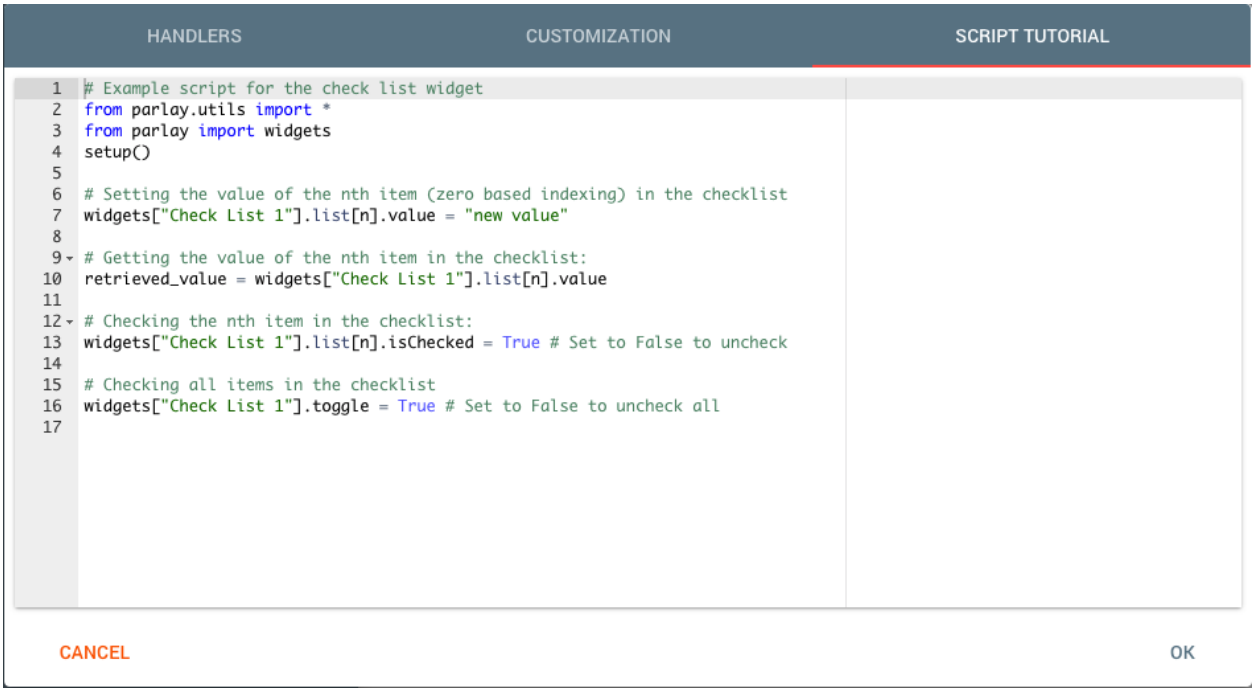
We've already created a button in our workspace, but if you haven't do so yet, add one to your workspace. Click the edit action in the widget's menu and head over to the customization tab. You should see the following tab:

Let's change the name of our button from "Button 1" to "My Renamed Button". Click OK to confirm to the changes and we should now see the changes reflected on the Button's title bar.

Widget Script Tutorial

Widgets that have properties bound to them should have a tab called the Script Tutorial. In this tab, there are quick tutorials on how to access and modify the properties of the currently edited widget in another widget's script.

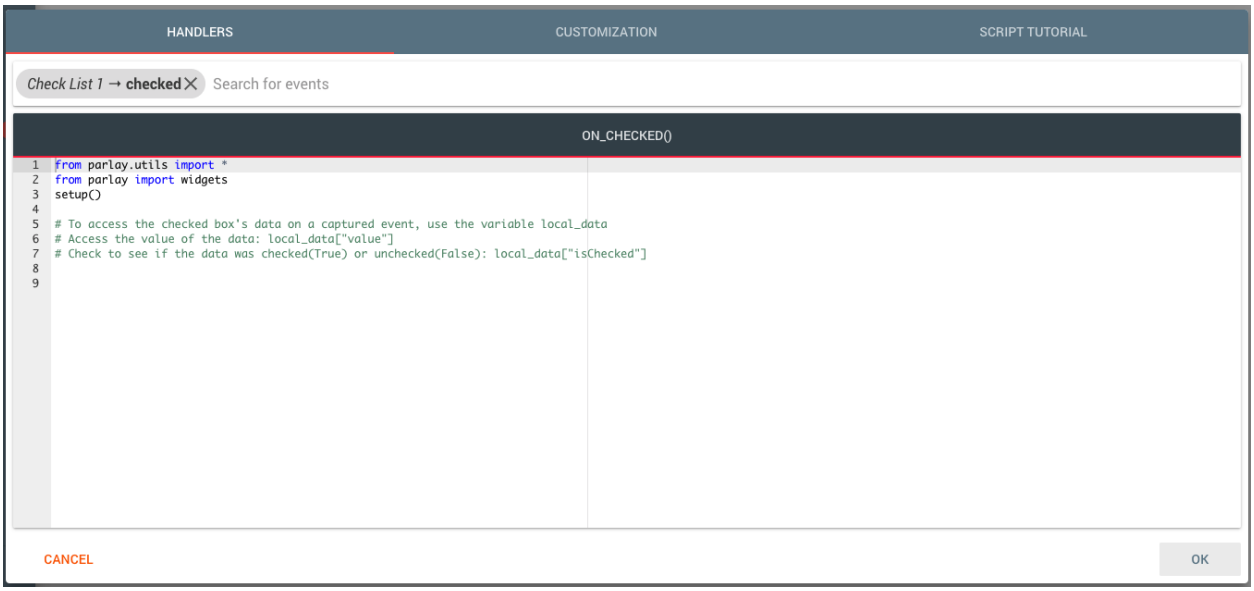
For example, if you add a checklist to your workspace, open the editing menu and move over to the Script Tutorial tab, you will see how you could modify or read that checklist's data from another widget.



The great thing about this script tutorial tab is that if you rename the widget, all instances of the old name in the tutorial become replaced with the new name. This makes scripting easier, as you can now just copy and paste the script helper in the tutorial tab!

Widget Local Data Injection

Parlay Widgets have support for custom events. These custom events have data bound to them, stored in a Python variable named "local_data". Widgets that have support for these local data injections will include a tutorial on how to properly use the local_data variable in their script in the Event Handler tab. Return to the checklist we just created and move back to the Event Handler tab. You should see a tutorial on all the possible ways you can use the local_data variable in this event handler's script.



Read the commented section on how to use the local_data variable. Now add the following line to the script:

```
print local_data["value"]
```

Now fill in the checklist with the following data:

- 111
- 222
- 333
- 444
- 555

Proceed with checking any of the items in the checklist. Did you notice that the checked item was printed to the screen?

2.4.3 Widget Examples

There are three main functional ways to use widgets:

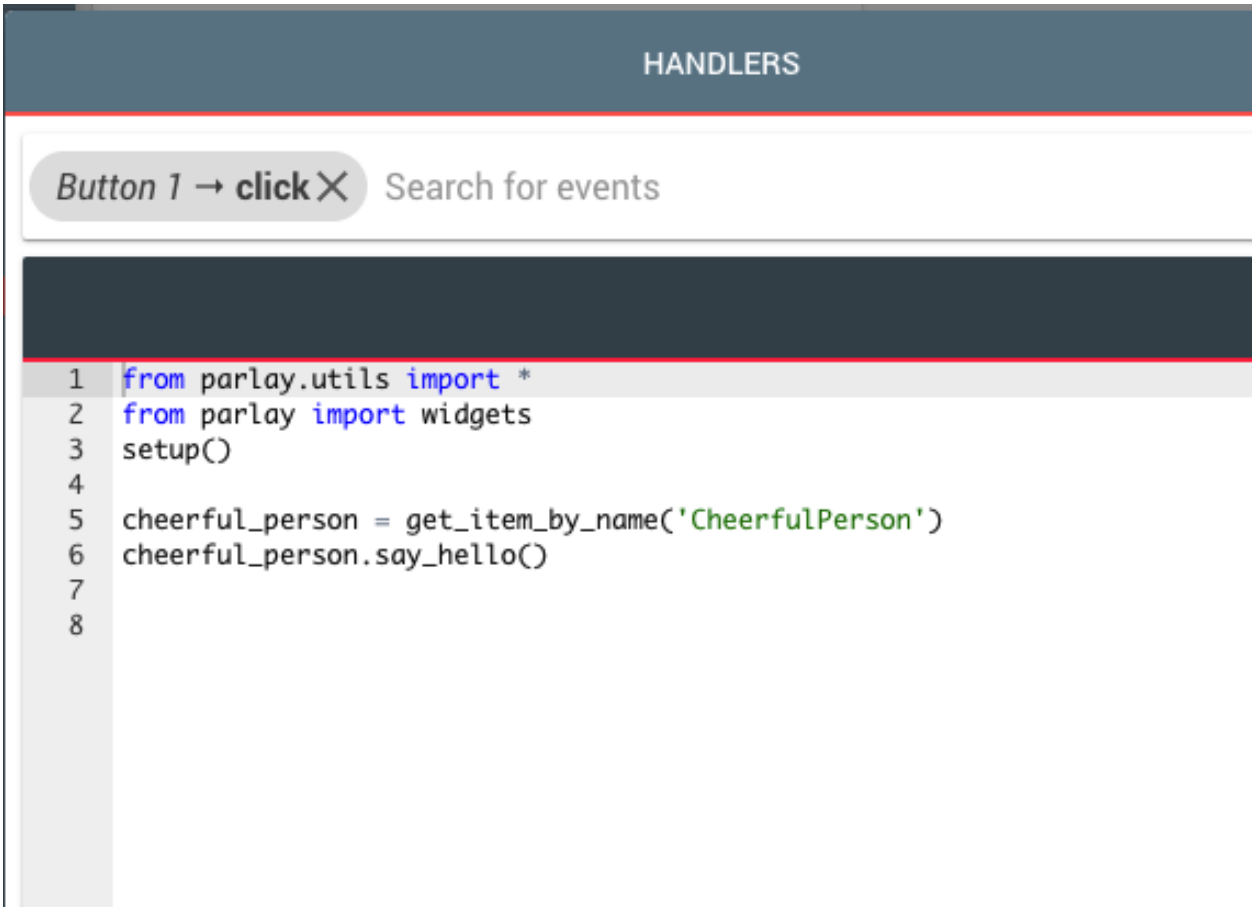
- Send queries or commands to Parlay Items
- Display Data from Parlay Items
- Modify or read data from other widgets

Let's review an example of each one

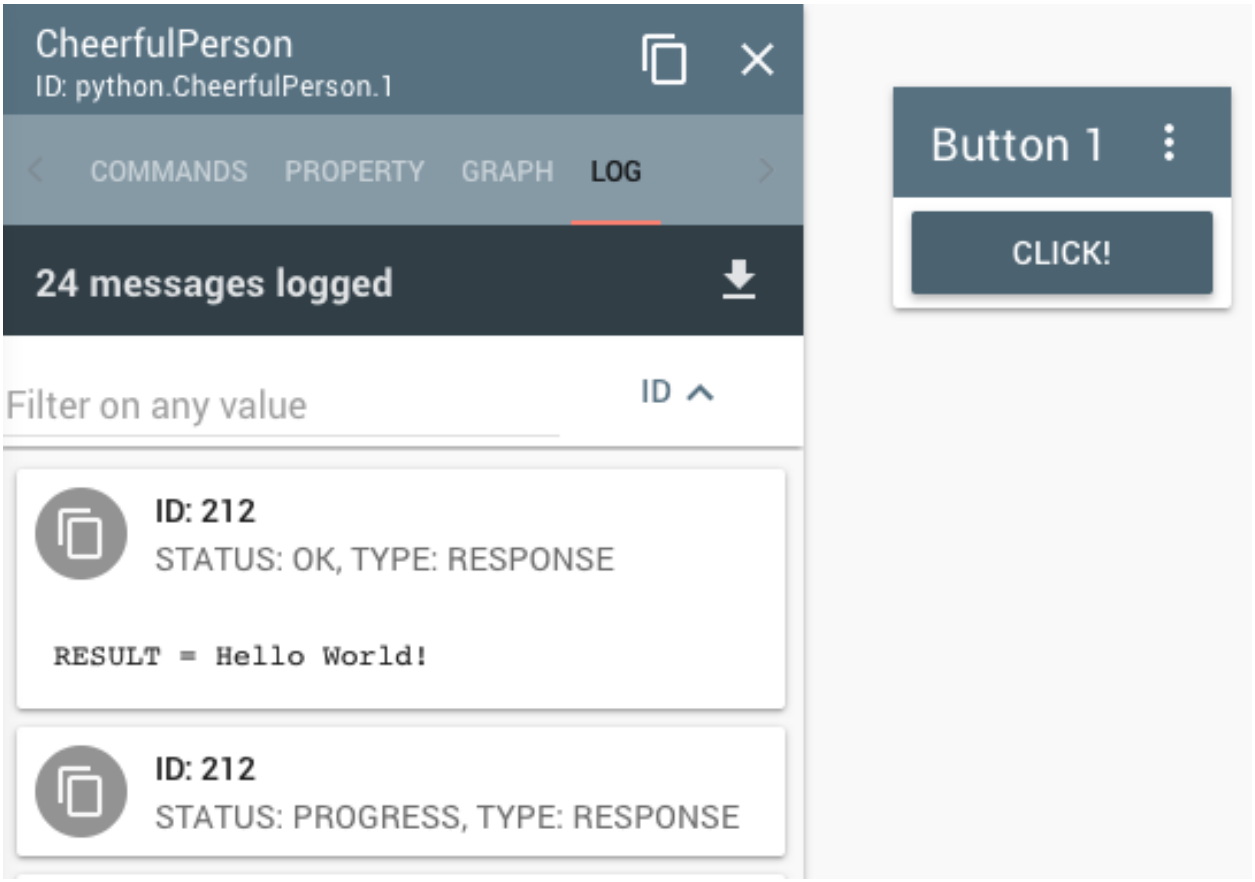
Interacting with Parlay Items

For this example, we will be using the CheerfulPerson item that we created in the hello world tutorial. Now create a button. Open the edit screen and go to the Event Handler tab. After confirming that the "click" event is selected, type in the following code:

```
cheerful_person = get_item_by_name("CheerfulPerson")
cheerful_person.say_hello()
```



Confirm the changes by clicking OK and then click the button. You won't see a noticeable change, but if we head over to the logs of out CheerfulPerson item, we can see messages being passed from the Parlay Server to the interface every time we click our newly created button!



Display Data from Parlay Items

Review the *Widget Transformer* tab on this page! Using the transformer tab as directed on this tutorial will correctly display data from Parlay Items onto your display widgets!

Interacting with Other Widgets

Recall from the *Widget Script Tutorial* that widgets that have properties can be modified by other widgets. Also recall that each event handler has three lines prepended to the script that should never be deleted. One of those lines imports a variable called "widgets" which is a collection of all created widgets in our workspace. Each widget can be modified or read by any Python Script that is run within the UI.

For this example we will use two buttons and a checklist. For the ease and clarity of this exercise, name the first button "check/uncheck random item" and the second "set random item". Name the checklist "list" and leave all items blank.

In our "check/uncheck random item" button, after ensuring the "click" event is attached, type the following code below the existing lines:

```
from random import randrange

# get the checklist we want to modify
checklist = widgets["list"]
# get a random number 0-4
index_to_change = randrange(5)
```

(continues on next page)

(continued from previous page)

```
# get the value of the checklist we want to modify
isChecked = not checklist.list[index_to_change].isChecked
# reverse the value of the checked
checklist.list[index_to_change].isChecked = not isChecked
```

In our "set random item" button, add the following code:

```
from random import randrange

# get the checklist we want to modify
checklist = widgets["list"]
# get a random number 0-4
index_to_change = randrange(5)
# get a random number 0-100
random_val = randrange(101)

# set random value to the value
checklist.list[index_to_change].value = random_val
```

Now click both buttons and see how it modifies our list. You should see that random items are getting checked and unchecked and that the items input fields are being replaced with random numbers!

2.4.4 Discovery

On the left Side Navigator, there is a button that says "Discovery". Clicking this button will request a discovery to the Parlay Connect server. Note that you MUST be connected to Parlay Connect for this to properly work.

2.4.5 Workspaces

You may have noticed that by adding a few widgets and then later refreshing your page results in losing all widgets in your workspace! Fear not, this doesn't always have to happen. Right underneath the discovery button in the left side navigator, you should see another button that says "workspaces" on it. Click it, and a dialog like the one below should appear:

Saving and Clearing the Current Workspace

Try adding a few widgets and items, adding some small scripts to some of the widgets and reorganizing the placement of the widgets via the drag and drop functionality. Now reopen the workspaces dialog. You should see "Current Workspace" with the number of widgets you just added directly beneath it. To the right you should see two buttons: "save as" and "clear". By clicking "save as", you will be prompted for a name for the workspace that you are about to save. Workspace saving is extremely useful as it saves all widgets and items in your workspace as well as well as other meta data attached to it. Meta data attached to widgets includes the position of the widget or item, scripts attached to

the widget, streams enabled, and other data filled in the input fields (for widgets like tables and check lists). Clicking clear is pretty self explanatory (and dangerous! use with caution). Once clear is clicked all items and widgets in your workspace are removed. The only way to bring back widgets cleared is if you already have saved the workspace prior to clearing it.

Autosave

If you accidentally cleared the workspace but forgot to save it, there's a good chance that the auto save discovery saved a partial or complete state of your cleared workspace. If an auto saved discovery is available it will show in the workspace dialog as seen below:

Managing Saved Workspaces

Any saved workspace (or autosaved workspace) will have a "load" button and a "delete" button to the right hand side of its entry. Clicking load will clear the current workspace and replace it with the workspace that you are loading. Delete will remove the saved workspace from Parlay. You will no longer have access to the deleted workspace, so take caution when deleting workspaces you no longer need.

At the bottom of the workspace dialog there are three final buttons: "Clear", "Export" and "Import". The clear button clears all saved workspaces. When clicking this button, you will be warned that this action cannot be undone.

Exporting and Importing Workspaces

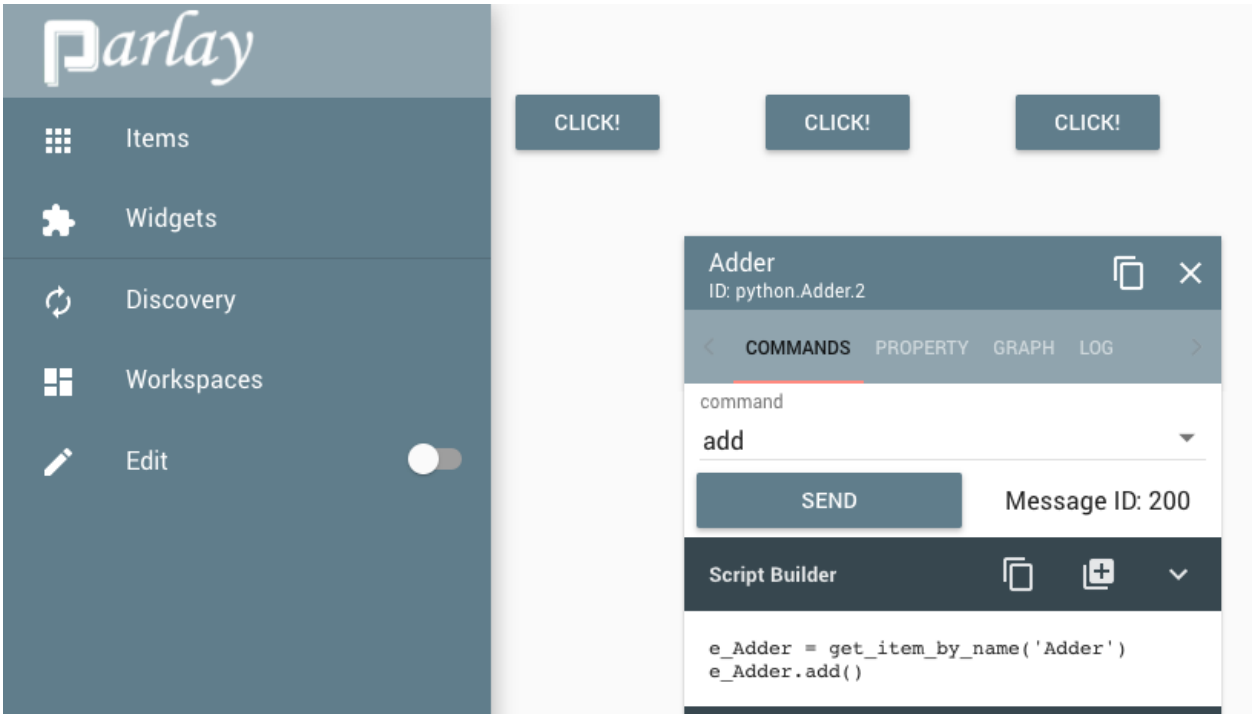
The export and import functions of the workspace is a very useful tool to be able to collaboratively work on projects with other team mates, co-workers, class mates or colleagues. Clicking the export button will prompt you to download a text file to your local system. This exported file contains all of your saved workspaces (excluding the autosaved workspace). You can now send this file to a colleague, and they can import the file so they may work with the same workspace that you just created.

In order to import a workspace, you must first click the import button. Doing so will prompt you to select a text file in your hard drive containing the data contents of the workspace. After loading the workspace (this may take a few econds to complete) you should see new workspaces made available for you to load onto your screen.

2.4.6 Editing Mode

On the left side navigator there is a toggle switch next to the "Edit" label. Clicking this switch will toggle the edit mode on your screen. You should notice that when the toggle is off, all of the title bars on each of the widgets disappear. As

long as edit mode is off, you will not be able to add widgets or items, nor will you be able to drag and drop existing widgets. Furthermore, an item card's title bar buttons will also be disabled.



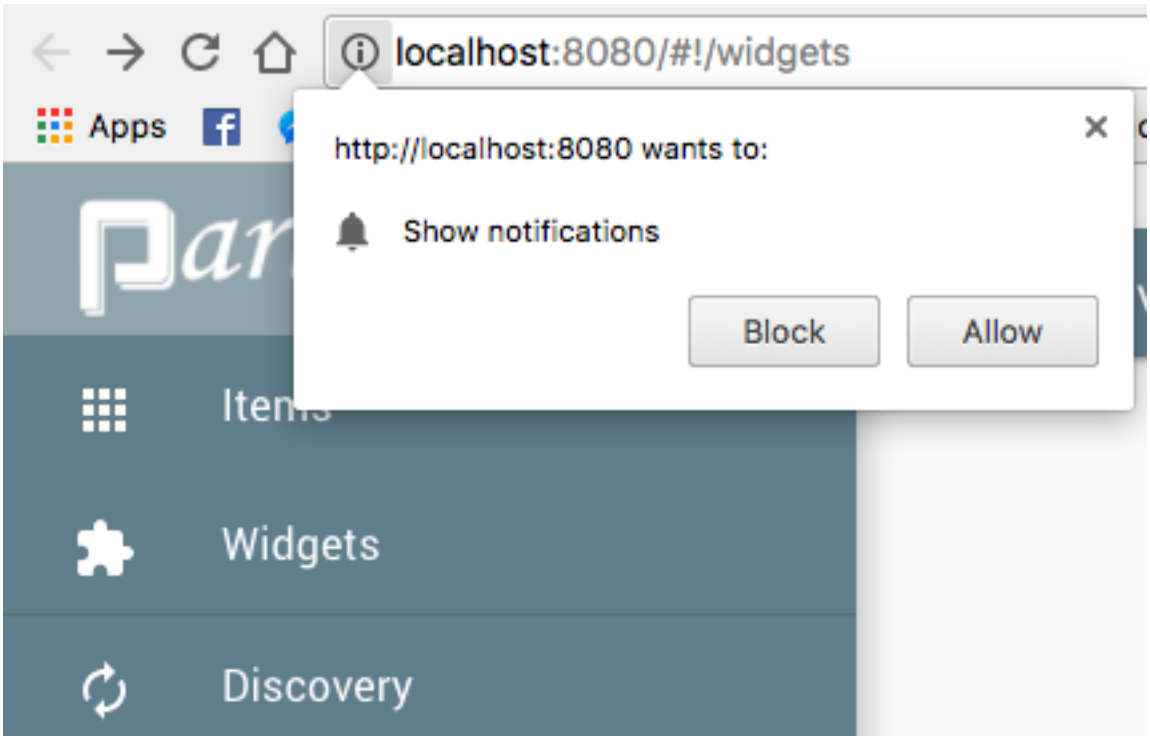
2.4.7 Protocols

There are often times, especially for embedded programmers and developers, where Python Parlay items are not sufficient to run all of your testing tools. Parlay has support for other protocols (such as USB, Bluetooth, etc) to be connected to the system so you may directly communicate to your embedded devices from the Parlay System.

Near the bottom of the left side navigator, you should see a button that says "Protocols". Clicking this will prompt open a dialog that will allow you to connect to different protocols available in your Parlay System. By connecting to your specified protocol, you will then be able to request a discovery for items connected in that protocol.

2.4.8 Notifications

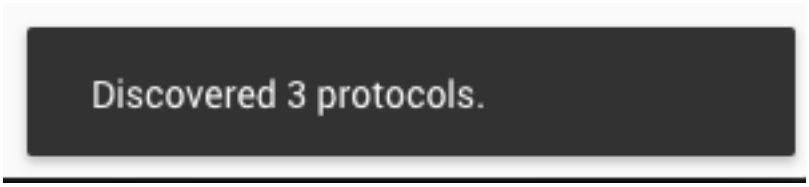
ParlayUI is a system that works with notifications. The first time you launch the Parlay system, Parlay will ask if you would like to enable notifications.



It is not necessary that you allow these notifications. With enabled notifications, your browser will notify you whenever a Parlay Notification occurs AND you are not currently viewing the Parlay Tab in your browser. If you disable notifications, you will still be able to view them from within the parlay system.

If you look near the bottom of the left side navigator, underneath the Protocols button, you should see notifications. Clicking this tab will pull open a side navigator on the right hand side of the window with a history of all of the Parlay notifications you have received in your current session.

You will also see notifications come up on your screen in real time in the bottom right corner of the window as a toast notification.



2.4.9 Settings

The last entry in the left side navigator is the settings button. Clicking this will open the settings configuration window where you can modify the different aspects of your Parlay System. Most of the settings can be explained by the labels within each entry in the window, but there is one important feature that should be noticed.

At the top of the Settings window, you should an entry for discovery settings. There is an option to disable automatic discovery on the start of the Parlay System. You may be wondering, why would I want to do this? This can be essentially crucial for users who are dealing with large network of embedded systems. Often times, in large networks of Parlay Items and embedded devices, Discovery can take minutes to complete. By disabling the auto discovery, you can continue to work by simply loading a saved discovery file, and having Parlay read the file and load discovery from the file rather than wait to find every single available item in a discovery request.

This does imply that you will have to run one discovery before being able to save and load the file. Once your discovery is complete, you can open the settings window and click the "Save Discovery" which will prompt a download of a text file containing the response contents of the Discovery Request. Now in the future, instead of having to wait minutes for a discovery to load, you can just click the "Load Discovery" button and have Parlay load the discovery in a much quicker time lapse.

2.5 Introduction to Scripting

Parlay scripts are extremely simple. By import a few functions from parlay, you can repeatably and automatically control any item that is connected to Parlay.

2.5.1 Scripts connect to Parlay

An important thing to understand about Parlay scripts is that they run separately from the Parlay system. To run a script that controls items that are connected to Parlay, *first run Parlay*, in the same way as the hello_world tutorial. Then, in a separate command line window, run your script. The script will automatically connect to the Parlay broker over a websocket connection. It can then send messages to and receive messages from other items that are connected to the Parlay broker.

2.5.2 Basic Script to control Hello World item

Let's start with a very simple script, that will send a command to, and read the response from, the "CheerfulPerson" item that we created in the hello_world tutorial.

```
# simple_script.py

from parlay.scripts import setup, discover, get_item_by_name

setup()  # required at the beginning of every script

print "Discovering..."
discover()  # required for future calls to 'get_item_by_name' to succeed
print "Discovery Complete!"

# cheeful_person is now the object representing the "CheerfulPerson" item connected␣
↪to Parlay
cheerful_person = get_item_by_name("CheerfulPerson")

# send a command to cheerful_person just by calling the function corresponding to the␣
↪command
# response is the data returned by the command function in the item
response = cheerful_person.say_hello()

print response
```

First, run the hello_world example, and *leave it running*. You can close the browser user interface window if you wish (it doesn't matter, because the Parlay server is still running in the command line window).

Then, run this script in a separate command line from the hello world example, and you should see the following output:

```
c:\> python simple_script.py

 Discovering...
 Running discovery...
 Discovery Complete!
 Hello World!
```

2.5.3 Elements of a script

These are the basic functions you can call in a script. To use any of them, import them from parlay.scripts, as seen in the simple example above.

`setup()` MUST be called at the beginning of every script.

`discover()` is almost always required, since it will allow future calls to `get_item_by_name` to succeed.

`get_item_by_name("NAME HERE")` returns an object that represents an item that is connected to the Parlay server. This will fail if the item is not connected.

During discovery, any connected items will also report what commands they support. In the case of hello_world, the `CheerfulPerson` item supports the `say_hello()` command. In the script, all you have to do is call the function `say_hello()`. Behind the scenes, this will send a command to the item, and wait for the response to come back. This is known as a *blocking* call. The script will pause at this line until the item responds to the command. This is easy to understand for beginners, and is a very common use case for scripts.

2.5.4 Serial vs Parallel commands

Scripts are run in a separate process than the Parlay broker. All the function calls in scripts can block without disturbing the Parlay system. Command functions, like `say_hello()` that we called above, have been designed as blocking function calls. The script will pause at that function call until the item returns a response to the command. In our simple Hello World example, the response comes very quickly. However, Parlay is designed to work well with items that take a long time to complete their commands.

Suppose you are building an embedded device with several motors. Each of the motors might be represented as an item in Parlay, and you want to be able to move two motors at the same time. You don't want to have to send a "move" command to one motor and wait for it to be complete before sending a "move" command to a second motor.

The `send_parlay_command` function gives you a way to do that.

Send a command now, wait for response later

Below is a modification to the above example that uses `send_parlay_command`, which returns a handle to the command. You can use that handle later to wait for the response to the command.

```
cheerful_person = get_item_by_name("CheerfulPerson")
command_handle = cheerful_person.send_parlay_command("say_hello")
# other code can go here, which will execute without waiting for the response to "say_
↪hello"
response = command_handle.wait_for_complete()
```

Example with serial and parallel commands

To demonstrate, let's run a more complicated example, with two items that have slooooooooowwwww commands, and a script that exercises those commands both serially and in parallel.

```
# items_with_slow_commands.py

from parlay import start, local_item, ParlayCommandItem, parlay_command

@local_item()
class Item1(ParlayCommandItem):

    @parlay_command()
    def slow_command_1(self):
        self.sleep(5)
        return "Command 1 Completed!"


@local_item()
class Item2(ParlayCommandItem):

    @parlay_command()
    def slow_command_2(self):
        self.sleep(5)
        return "Command 2 Completed!"


if __name__ == "__main__":
    Item1(1, "Item 1")  # IDs can be integers or strings, but names must be strings
    Item2(2, "Item 2")  # You can create more than one instance of an item, as long␣
↪as they have distinct IDs
    start(open_browser=False)  # you can avoid launching your web browser
```

Run the previous file on the command line to start Parlay:

```
c:\> python items_with_slow_commands.py
```

In a separate command line, launch the following script:

```
# serial_vs_parallel_script.py

from parlay.scripts import setup, discover, get_item_by_name

setup()
discover()

item1 = get_item_by_name("Item1")
item2 = get_item_by_name("Item2")

print "\nSending blocking commands"

print "  Slow Command 1..."
response1 = item1.slow_command_1()
print response1

print "  Slow Command 2..."
response2 = item2.slow_command_2()
print response2
```

(continues on next page)

(continued from previous page)

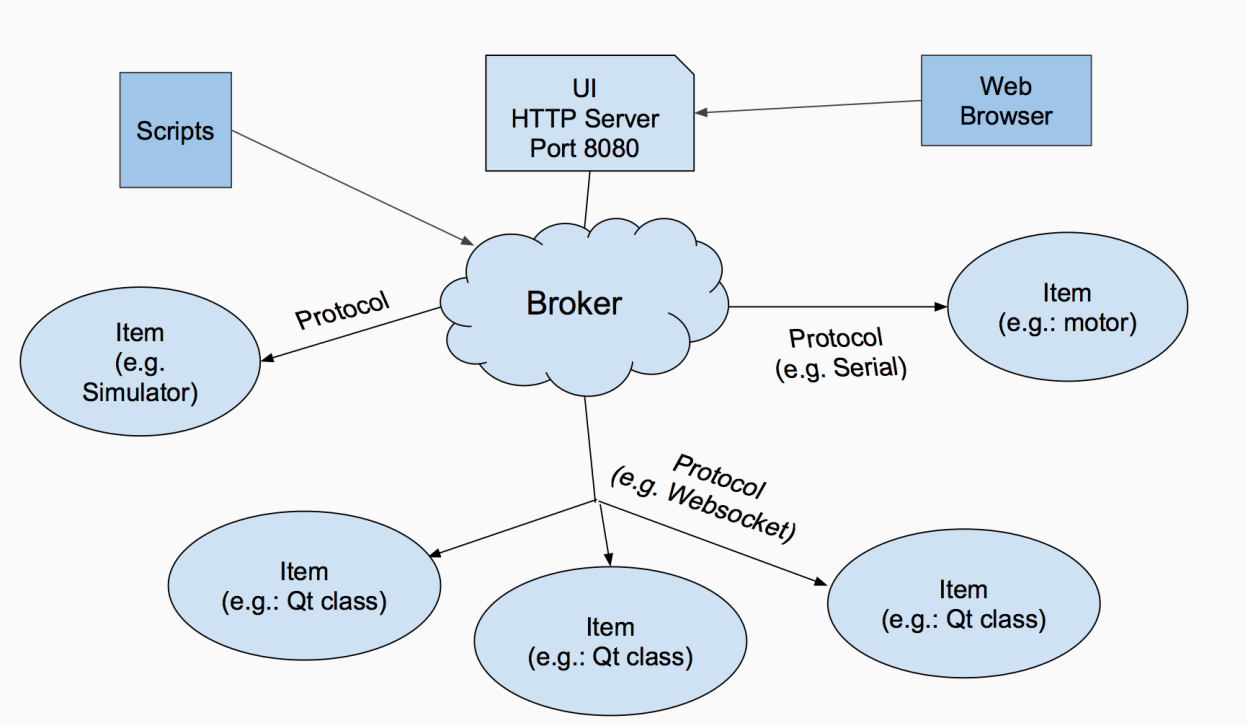
```
print "\nSending parallel commands"

print "  Slow Command 1..."
cmd1 = item1.send_parlay_command("slow_command_1")
print "  Slow Command 2..."
cmd2 = item2.send_parlay_command("slow_command_2")

print "  Waiting for responses..."
response1 = cmd1.wait_for_complete()
response2 = cmd2.wait_for_complete()

print response1
print response2
```

```
c:\> python serial_vs_parallel_script.py
Connecting to localhost : 8085
Running discovery...

Sending blocking commands
  Slow Command 1...            <--- this takes 5 seconds
Command 1 Completed!
  Slow Command 2...            <--- this takes 5 seconds
Command 2 Completed!

Sending parallel commands
  Slow Command 1...
  Slow Command 2...
  Waiting for responses...     <--- this takes 5 seconds
Command 1 Completed!
Command 2 Completed!
```

As expected, running the two commands serially takes about 10 seconds, while running them in parallel takes only 5 seconds.

You can mix and match serial and parallel commands to any level of complexity, which enables very powerful scripting capabilities.

2.6 Overview of Parlay Architecture



2.6.1 Core Concepts

Items

In Parlay, an Item represents something physical (e.g. motor) or logical (e.g. scheduler). Every item has an 'ID' and a 'Name'.

The ID determines the Item's identity, and must be unique among all items connected to the Parlay broker. It can be a string or a number. The broker uses the Item's ID to correctly route messages.

In the most general case, an Item is an object that can send and receive messages that are routed by the Parlay broker. However, when creating your own Items, you will inherit from Parlay Item classes that provide more functionality out of the box.

Parlay provides two standard components for Items that are supported by the built-in User Interface and by Parlay scripts:

- Commands: Take arguments and return a result or perform an action
- Properties: - Can be "GET" and "SET" dynamically - Can be streamed at a specified rate, or whenever they change

Discovery

One of the most powerful features of Parlay is discovery: finding out what items are connected (and how they are connected) in real time, rather than hard-coding them up-front. For example, you can write a script that sends various commands to a serial motor controller item with the ID of "Motor1". You can plug that motor controller into any COM

port on your PC, or even write a simulator in python that simulates the communication with that motor controller. As long as the item returns the ID "Motor1" during the discovery operation, your script will work with zero changes.

The Broker

At the core of Parlay is the Broker. It is a publish/subscribe message broker that allows all items to send messages to each other.

The broker automatically provides a websocket server that listens for incoming connections from the various Parlay integration libraries (.Net, Qt, JS, etc.)

The broker also provides a webserver that serves the built-in browser-based user interface.

Messages

Commands and responses, getting and setting properties, and asynchronous events are all implemented as JSON messages that are routed through the broker.

Example Command message from a script to an Item:

```
{
    "TOPICS": {
        "FROM": "script.simple_script.py",
        "TO": "CheerfulPerson",
        "MSG_ID": 100,
        "MSG_TYPE": "COMMAND",
        "RESPONSE_REQ": True,
        "TX_TYPE": "DIRECT"}
    },

    "CONTENTS": {
        "COMMAND": "say_hello"
    }
}
```

Protocols

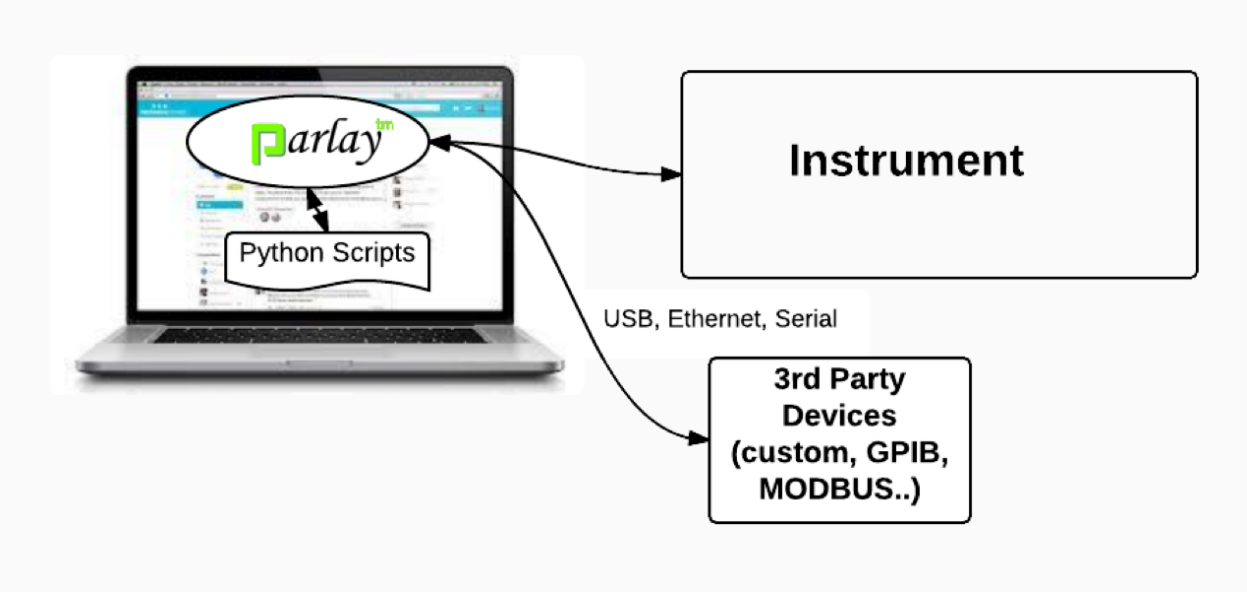
The broker supports custom protocols written in Python that can integrate with components that do not natively speak the Parlay protocol. Got an off-the-shelf USB camera or serial motor controller that you want to hook up to Parlay? Write a custom protocol for it (see how below) and it will be connected to the Parlay system, and can be controlled by the browser UI, scripts, and any other item connected to the system.

2.6.2 Two ways of running Parlay

Parlay is cross-platform, and it is designed to run either on a PC or an embedded system.

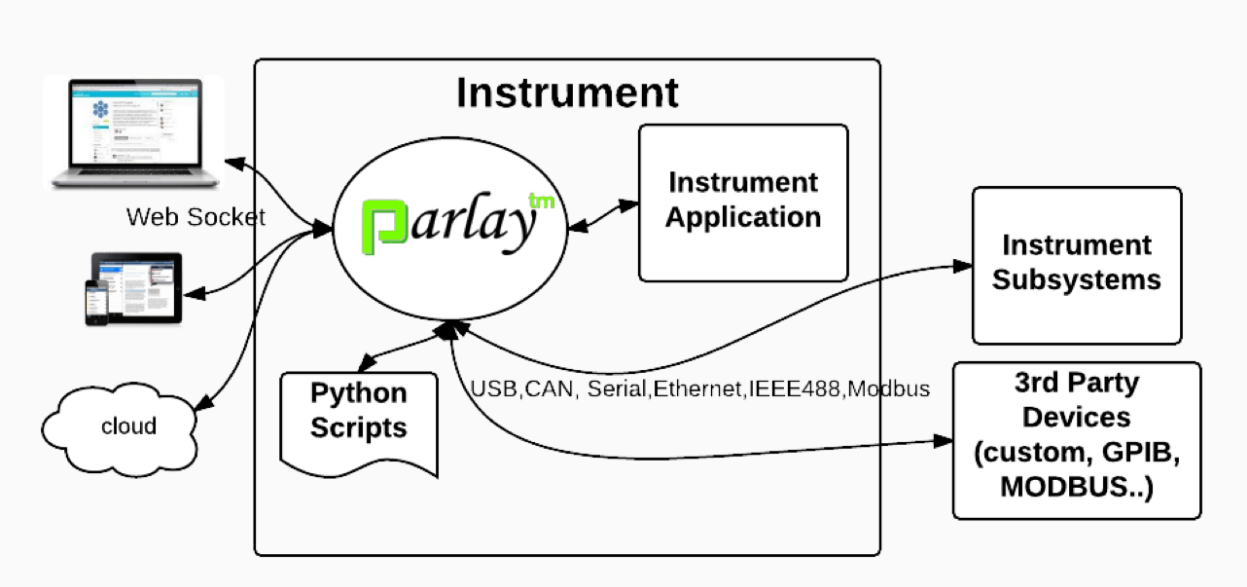
PC Rapid Prototyping Mode

Parlay can be installed on your PC to aid development. This is how you would most likely work with Parlay during the early stages of product development.



Device Embedded Mode

Parlay can also be installed on any embedded device that runs an operating system that supports Python, such as any version of Linux or Windows. Parlay is designed to be incorporated as a key component of your product. It provides powerful service and diagnostic capabilities to production devices.



2.7 Custom Protocols

The Parlay Broker supports custom protocols written in Python that can integrate with any off-the-shelf or custom components that communicate any some way.

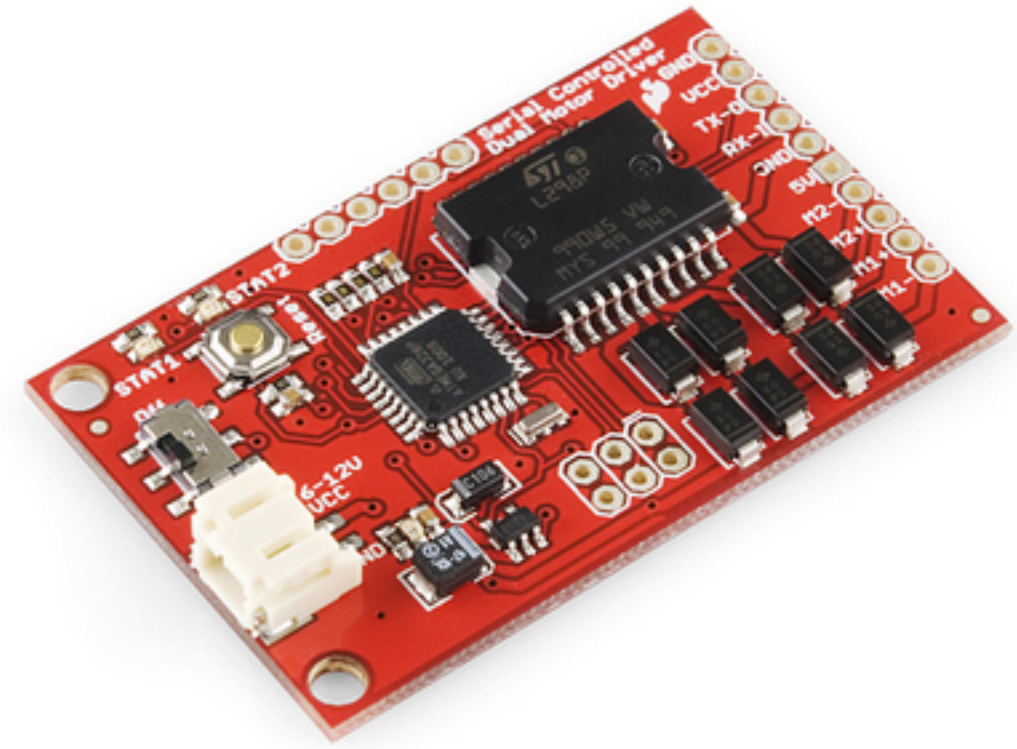
Got an off-the-shelf USB camera or serial motor controller that you want to hook up to Parlay? Write a custom protocol for it (see how below) and it will be connected to the Parlay system, and can be controlled by the browser UI,

scripts, and any other item connected to the system.

The best way to demonstrate is with an example.

2.7.1 Case Study: Serial DC Motor Controller

This tutorial will walk you through connecting Parlay to the Serial Controlled Motor Driver from Sparkfun. Details and specifications of the motor controller can be found at https://www.sparkfun.com/products/9571.



Motor Controller Serial Interface

The interface with the motor controller is very simple. The motor controller drives two DC motors. Over a serial connection, we can command the motors to spin with an ASCII string formatted like "1f5\r". There are three characters in the command, and then it is terminated with a carriage return (not a newline character!).

- 1st character: "1" means motor 1, "2" means motor 2
- 2nd character: "f" means spin forward, "r" means spin in reverse
- 3rd character: "5" means spin at speed 5, in range 0-9. 0 is stopped, 9 is 100% PWM.

The parameters for the serial interface are 115200 baud, 8-N-1.

Desired Commands

Since we are writing a custom Protocol and custom Item to interact with this motor controller, we can decide how we want our users to interact with our Item. The users don't have to know the serial interface described above – they can interact with the motors using commands that we create. So, let's decide on the following requirements:

- We want Motor 1 and Motor 2 to be separate Items
- We want to have understandable commands for our motors: "Spin" and "Stop"
- When we give our motor a "Spin" command, we want to be able to specify the speed as a positive or negative number
- These commands should be translated into the correct serial string to be sent to the motor controller

The code

Here is the code to achieve this. We will examine it a bit at a time.

```
from parlay.protocols.serial_line import ASCIILineProtocol, LineItem
from parlay import parlay_command, start

class SerialMotorControllerProtocol(ASCIILineProtocol):
    def __init__(self, port):
        super(SerialMotorControllerProtocol, self).__init__(port=port)
        self.items = [SerialMotorControllerItem(1, "Motor1", "Motor1", self),
                      SerialMotorControllerItem(2, "Motor2", "Motor2", self)]

    @classmethod
    def open(cls, broker, port="/dev/ttyUSB0"):
        return super(SerialMotorControllerProtocol, cls).open(broker, port=port,␣
↪baudrate=115200, delimiter="\r")


class SerialMotorControllerItem(LineItem):
    def __init__(self, motor_index, item_id, name, protocol):
        LineItem.__init__(self, item_id, name, protocol)
        self._motor_index = motor_index

    @parlay_command()
    def spin(self, speed):
        """
        Move the motor at a constant speed, between -9 and 9.  Negative speed causes␣
↪motor to spin in reverse.
        :param speed: speed to move
        :type speed int
        :return: serial response from motor controller
        """

        speed = int(speed)
        if speed > 9 or speed < -9:
            raise ValueError("Speed outside range")  # this exception causes an error␣
↪message to be sent back to whoever sent the command
        direction = "f" if speed >= 0 else "r"
        self.send_raw_data("{}{}{}".format(self._motor_index, direction, abs(speed)))

        # this waits for a response string from the motor controller, which will be␣
↪shown in the UI as "result"
```

(continues on next page)

(continued from previous page)

```
        return self.wait_for_data()

    @parlay_command()
    def stop(self):
        """
        Stop the motor
        :return: serial response from motor controller
        """
        # returns what spin() returns, which is the motor controller response string
        return self.spin(0)


if __name__ == "__main__":

    # any protocol that has been imported or defined will be available to open
    #  so we do not need to instantiate anything before calling start()

    start()
```

The Protocol Class

The first step is to create our own Protocol class, which we will call `SerialMotorControllerProtocol`.

```
class SerialMotorControllerProtocol(ASCIILineProtocol):
```

Parlay has pre-built Protocol classes for many common interfaces, including delimited ASCII serial communication. The class that does this is `ASCIILineProtocol`, which will communicate over a serial line using our specified COM port, baudrate, and delimiter character.

Override the `__init__` method

To fulfill our requirements that we specified above, we must override the `__init__` function and populate the `self.items` list with our `SerialMotorControllerItem` objects (described below). The items in `self.items` will be visible after the user has performed a discovery.

```
def __init__(self, port):
    super(SerialMotorControllerProtocol, self).__init__(self, port=port)
    self.items = [SerialMotorControllerItem(1, "Motor1", "Motor1", self),
                  SerialMotorControllerItem(2, "Motor2", "Motor2", self)]
```

Override the `open` class method

`SerialMotorControllerProtocol` inherits from `ASCIILineProtocol`, which inherits from `BaseProtocol`. `BaseProtocol` has an `open` method that any child class *must* override. `ASCIILineProtocol` already does this, which handles setting up the serial port with the desired settings.

For our motor controller, the baudrate and delimiter character are specified by the hardware, so there's no need to make the user specify that. So, in `SerialMotorControllerProtocol`, we also override the `open` class method and specify the baudrate to be 115200 baud, and the delimiter character to be "\r", or carriage return.

The `broker` argument of the `__open__` function is required.

```
@classmethod
def open(cls, broker, port="/dev/ttyUSB0"):
    return super(SerialMotorControllerProtocol, cls).open(broker, port=port,␣
↪baudrate=115200, delimiter="\r")
```

When calling our parent's `open` method, we must use python's `super` function like so: `super(SerialMotorControllerProtocol, cls).open(...)`

If we were to override `open` like below, our protocol would be shown in the Parlay User Interface as a `ASCIILineProtocol`, rather than a `SerialMotorControllerProtocol` like we want:

```
@classmethod
def open(cls, broker, port="/dev/ttyUSB0"):
    # WRONG! DON'T DO THIS!
    return ASCIILineProtocol.open(broker, port=port, baudrate=115200, delimiter="\r")
```

Using the base class `get_discovery` method

`SerialMotorControllerProtocol` inherits from `ASCIILineProtocol`, which inherits from `BaseProtocol`. `BaseProtocol` has a `get_discovery` method defined as follows:

```
def get_discovery(self):
    return {'TEMPLATE': 'Protocol',
            'NAME': str(self),
            'protocol_type': getattr(self, "_protocol_type_name", "UNKNOWN"),
            'CHILDREN': [x.get_discovery() for x in self.items]}
```

For the base `get_discovery` method to work, the protocol must create a list of items in `self.items` that each support their own `get_discovery` method.

We already took care of this in our `__init__` method as described above. As described below, our `SerialMotorControllerItem` class inherits from `ParlayCommandItem`, which means that it supports the `get_discovery` method.

The Item

The second step is to create our own Item class, which we will call `SerialMotorControllerItem`. It inherits from `LineItem`, which is a pre-built class designed to work with a serial protocol. It provides the helper function `send_raw_data`, which we will use later to send our commands out the serial port.

The `__init__` method

We must call our parent's **init** function (not necessary to use `super` here). We also store the provided `motor_index` in a member variable so we can correctly format the command strings to be sent over the serial port to the motor controller.

```
class SerialMotorControllerItem(LineItem):
    def __init__(self, motor_index, item_id, name, protocol):
        LineItem.__init__(self, item_id, name, protocol)
        self._motor_index = motor_index
```

The `spin` command

`SerialMotorControllerItem` inherits from `LineItem`, which inherits from `ParlayCommandItem`. This base class takes care of a lot of grunt work for you to make your command functions be discoverable and visible in the Parlay User Interface.

To make a command that is visible in the UI, just create a function and decorate it with `@parlay_command`.

```
@parlay_command()
def spin(self, speed):
    """
    Move the motor at a constant speed, between -9 and 9.  Negative speed causes␣
↪motor to spin in reverse.
    :param speed: speed to move
    :type speed int   # the Parlay UI can use type hinting to force the user to enter␣
↪an integer
    :return: serial response from motor controller
    """
    speed = int(speed)
    if speed > 9 or speed < -9:
        raise ValueError("Speed outside range")  # this exception causes an error␣
↪message to be sent back to whoever sent the command
    direction = "f" if speed >= 0 else "r"
    self.send_raw_data("{}{}{}".format(self._motor_index, direction, abs(speed)))

    # this waits for a response string from the motor controller, which will be shown␣
↪in the UI as "result"
    return self.wait_for_data()
```

The `stop` command

Stopping the motor is just sending it a `spin` command with speed = 0. We can do that! Once again, we add the `@parlay_command` decorator to the function.

```
@parlay_command()
def stop(self):
    """
    Stop the motor
    :return: serial response from motor controller
    """
    # returns what spin() returns, which is the motor controller response string
    return self.spin(0)
```

Starting Parlay

If this file is called as a python script, such as `$ python motor_controller.py`, we can start parlay automatically. Otherwise, we can import this file in any other python file to use the `SerialMotorControllerProtocol` and `SerialMotorControllerItem` that we have just defined.

```
if __name__ == "__main__":
    start()
```

2.8 Tutorials

Below are some tutorials that walk you through specific tasks you may want to accomplish with Parlay.

2.8.1 Logging

Parlay uses the standard python logging library to log messages, warnings and errors. All parlay JSON messages are logged at the DEBUG level. Warnings and Errors are logged at their respective levels.

Parlay uses the recommended logging schema where the package name of the module that is logging (e.g. parlay.server.broker). That means that if you want to log to a file, email, udp, etc you can add your handler to the "parlay" logger, or the root logger

Below is a simple example of how to attach a custom handler to the root logger that will cycle files every hour. See the python logging documentation for more information on attaching handlers, filtering messages and logging.

```
from parlay import start, local_item, parlay_command, ParlayCommandItem

@local_item()
class CheerfulPerson(ParlayCommandItem):

    @parlay_command()
    def say_hello(self):
        return "Hello World!"

if __name__ == "__main__":

    import logging
    from logging.handlers import TimedRotatingFileHandler

    # get the ROOT logger for the entire process
    logger = logging.getLogger()

    # add the file handler to the root logger. Now any logged messages will be logged␣
↪to files every hour (max 10)
    logger.addHandler(TimedRotatingFileHandler("LOG.txt", when="H", backupCount=10))

    # construct a CheerfulPerson item
    CheerfulPerson(1, "Doug")

    start()
```

The parlay logger defaults to logging.DEBUG . To change the default log level of the parlay logger you can pass log_level to start() :

```
start(log_level=logging.WARN)
```

You can also customize how Parlay prints its log messages. The snippet below enhances the example by adding a timestamp to the log file handler before adding it to the logger:

```
handler = TimedRotatingFileHandler("LOG.txt", when="H", backupCount=10)
formatter = logging.Formatter(fmt='%(asctime)s %(message)s')
handler.setFormatter(formatter)
logger.addHandler(handler)
```

2.9 Parlay API Documentation

This is the documentation for the complete Parlay API.

2.9.1 Parlay top level members

These functions and classes are defined in submodules, but are so commonly used that they are made available directly as members of the parlay module.

parlay.**open_protocol**(*protocol_name*, ***kwargs*)

Sets up a protocol to be opened after the Broker initializes.

This function has the same effect as opening a new protocol from the browser-based user interface.

Parameters

- **protocol_name** (*str*) – name of protocol class to call the open method
- **kwargs** – keyword arguments to pass to the protocol's _open_ method

Returns none

Example Usage:

```
from parlay import open_protocol, start
open_protocol("ASCIILineProtocol", port="/dev/ttyUSB0", baudrate=57600)
start()
```

parlay.**local_item**(*auto_connect=False*)

A class decorator for python Items that are not part of an external protocol.

Local items are self-contained, and do not communicate over external protocols, for example a serial port. They are typically used for simulators or pure python computation items.

Parameters **auto_connect** – whether to automatically connect to the Parlay broker when the item is created.

Returns decorator function

Example usage of local_item decorator:

```
# motor_sim.py

@local_item()
class MotorSimulator(ParlayCommandItem):

    def __init__(self, item_id, item_name):
        ...
```

Example usage of defined local item:

```
import parlay
from motor_sim import MotorSimulator

MotorSimulator("motor1", "motor 1")  # motor1 will be discoverable
parlay.start()
```

class parlay.**ParlayCommandItem**(*item_id=None*, *name=None*, *reactor=None*, *adapter=None*, *parents=None*)

This is a Parlay Item that defines functions that serve as commands, with arguments.

This class enables you to use the `parlay_command()` decorator over your command functions. Then, those functions will be available as commands that can be called from the user interface, scripts, or by other items.

Example: How to define a class as a ParlayCommandItem:

```
from parlay import local_item, ParlayCommandItem, parlay_command

@local_item()
class MotorSimulator(ParlayCommandItem):

    def __init__(self, item_id, item_name):
        self.coord = 0
        ParlayCommandItem.__init__(self, item_id, item_name)

    @parlay_command
    def move_to_coordinate(self, coordinate)
        self.coord = coordinate
```

Example: How to instantiate an item from the above definition:

```
import parlay
from motor_sim import MotorSimulator

MotorSimulator("motor1", "motor 1")  # motor1 will be discoverable
parlay.start()
```

Example: How to interact with the instantiated item from a Parlay script:

```
# script_move_motor.py

setup()
discover()
motor_sim = get_item_by_name("motor 1")
motor_sim.move_to_coordinate(500)
```

class parlay.**ParlayProperty**(*default=None*, *val_type=<type 'str'>*, *read_only=False*, *write_only=False*, *custom_read=None*, *custom_write=None*, *callback=<function <lambda>>*)

A convenience class for creating properties of ParlayCommandItems.

Example: How to define a property:

```
class MyItem(ParlayCommandItem):
    x = ParlayProperty(default=0, val_type=int)

    def __init__(self, item_id, item_name):
        ParlayCommandItem.__init__(self, item_id, item_name)
    ...
```

Example: How to access a property from a script:

```
setup()
discover()
my_item = get_item_by_name("MyItem")
original_value = my_item.x
my_item.x = 5
```

class parlay.**ParlayDatastream**(**args*, ***kwargs*)

Deprecated

parlay.**parlay_command**(*async=False*, *auto_type_cast=True*)
Make the decorated method a parlay_command.

Parameters

- **async** – If True, will run as a normal twisted async function call. If False, parlay will spawn a separate thread and run the function synchronously (Default false)
- **auto_type_cast** – If true, will search the function's docstring for type info about the arguments, and provide that information during discovery

parlay.**start**(*mode='DEVELOPER_MODE'*, *ssl_only=False*, *open_browser=True*, *http_port=8080*, *https_port=8081*, *websocket_port=8085*, *secure_websocket_port=8086*, *ui_path=None*, *log_level=10*, *ui_caching=False*)
Run the default Broker implementation. This call will not return.

2.9.2 Utils

These are the most common functions for scripts. They work equally well in Parlay Items.

parlay.utils.**setup**(*ip='localhost'*, *port=8085*, *timeout=3*)
Connect this script to the broker's websocket server.

Parameters

- **ip** – ip address of the broker websocket server
- **port** – port of the broker websocket server
- **timeout** – try for this long to connect to broker before giving up

Returns none

parlay.utils.**discover**(*force=True*)

parlay.utils.**get_item_by_name**(*item_name*)

parlay.utils.**get_item_by_id**(*item_id*)

parlay.utils.**sleep**(*time*)

2.9.3 Full Parlay Module Listing

This is the complete, recursive listing for the parlay module. It is not guaranteed to be nicely formatted, but it should be complete.

parlay

parlay package

Subpackages

parlay.items package

Submodules

parlay.items.base module

class parlay.items.base.**BaseItem**(*item_id*, *name*, *adapter=None*, *parents=None*)
Bases: object

The Base Item that all other Items should inherit from

add_child(*child*)
Adds child to the children list of our item. NOTE: duplicates are not permitted.

Child should be an instance of a class that is derived from BaseItem.

Parameters **child** – BaseItem to add

Returns

get_discovery()
The protocol can call this to get discovery from me

get_item_template_string()
This returns the type string for the item eg: sscom/STD_ITEM “

is_child()
Determines if this item is a child of another item. Or in other words this Item has a parent. :return: boolean - True if this item is a child, False if not.

item_name = None
:type Adapter

on_message(*msg*)
Every time we get a message for us, this method will be called with it. Be sure to override this.

publish(*msg*)

subscribe(*_fn*, ***kwargs*)

exception parlay.items.base.**BaseItemError**(*item_id=None*)
Bases: exceptions.Exception

Exception used to indicate item construction failures. For example, if the __init__ function of a class inheriting from BaseItem is overridden and the parent’s __init__() function is not called.

class parlay.items.base.**INPUT_TYPES**
Bases: object

ARRAY = 'ARRAY'

BOOLEAN = 'BOOLEAN'

DROPDOWN = 'DROPDOWN'

NUMBER = 'NUMBER'

NUMBERS = 'NUMBERS'

OBJECT = 'OBJECT'

STRING = 'STRING'

STRINGS = 'STRINGS'

class parlay.items.base.**MSG_STATUS**
Bases: object

ERROR = 'ERROR'

INFO = 'INFO'

OK = 'OK'

PROGRESS = 'PROGRESS'

WARNING = 'WARNING'

class parlay.items.base.**MSG_TYPES**
Bases: object

COMMAND = 'COMMAND'

DATA = 'DATA'

EVENT = 'EVENT'

PROPERTY = 'PROPERTY'

RESPONSE = 'RESPONSE'

STREAM = 'STREAM'

class parlay.items.base.**TX_TYPES**
Bases: object

BROADCAST = 'BROADCAST'

DIRECT = 'DIRECT'

parlay.items.base.**get_recursive_base_list**(*cls*, *base_list=None*)
Get the full class heirarchy list for a class, to see *all* classes and super classes a python class inherits from

parlay.items.cloud_link module

class parlay.items.cloud_link.**CloudLink**(**args*, ***kwargs*)
Bases: *parlay.items.parlay_standard.ParlayCommandItem*

base_link_uri = 'https://pub1.parlay.cloud/channels'

connect_to_cloud_channel(***kwargs*)
Connect up to the cloud channel with UUID uuid :param uuid: the UUID of this device. The uuid determines which channel you are connected to :return:

connected_to_cloud = False

device_registration_uri = 'https://pub1.parlay.cloud/device_stats/api/v1/device'

disconnect_from_cloud(***kwargs*)

generate_keys(***kwargs*)

get_channel(***kwargs*)

get_public_key(***kwargs*)

http_request(***kwargs*)
Do an http request on the broker :type path str

register_device_with_cloud(***kwargs*)
Register the public and private keys with the cloud. If you already have them registered this will overwrite them :param username: username of a user with access to the cloud :param password: password of the user to authenticate with :param name: the name of the device :param serial: the serial number of the device :param group_id: the id given to the access group for this device :type group_id int :param notes: any notes to attach to this device :return: the UUID of the newly created device

uuid = None

class parlay.items.cloud_link.**CloudLinkSettings**
Bases: object

Class level settings to set for cloudLink

CLOUD_SERVER_ADDRESS = 'https://pub1.parlay.cloud'

PRIVATE_KEY_LOCATION = None

PRIVATE_KEY_PASSPHRASE = None

PUBLIC_KEY_LOCATION = None

UUID_LOCATION = None

class parlay.items.cloud_link.**CloudLinkWebsocketClient**(*adapter=None*)
Bases: autobahn.twisted.websocket.WebSocketClientProtocol

The websocket client that will bridge all messages between the broker and the cloud channel

connectionLost(*reason*)
Called when the connection is shut down.

Clear any circular references here, and any external references to this Protocol. The connection has been closed.

@type reason: L{twisted.python.failure.Failure}

onConnect(*response*)
Callback fired directly after WebSocket opening handshake when new WebSocket server connection was established.

Parameters **response** (instance of autobahn.websocket.protocol.ConnectionResponse) – WebSocket connection response information.

onMessage(*payload*, *isBinary*)
When we get a message from the cloud link, publish it on our broker PRIVATE_KEY_LOCATION ‘ :param payload: :param isBinary: :return:

send_message_to_cloud_channel(*msg*)
Send the message to the cloud. This should be subscribed to all messages :param msg: :return:

class parlay.items.cloud_link.**CloudLinkWebsocketClientFactory**(*cloud_item*, **args*, ***kwargs*)
Bases: autobahn.twisted.websocket.WebSocketClientFactory, twisted.internet.protocol.ReconnectingClientFactory

Websocket Client library that keeps track of the cloud item for the CloudLinkWebsocketClient

buildProtocol(*addr*)
Create an instance of a subclass of Protocol.

The returned instance will handle input on an incoming server connection, and an attribute "factory" pointing to the creating factory.

Alternatively, L{None} may be returned to immediately close the new connection.

Override this method to alter how Protocol instances get created.

@param addr: an object implementing L{twisted.internet.interfaces.IAddress}

disconnect()

class parlay.items.cloud_link.**CloudStressTest**(**args*, ***kwargs*)

Bases: *parlay.items.parlay_standard.ParlayCommandItem*

count_up(***kwargs*)

Parameters **max** –

:type max int :return:

data = 0.0

sleep_time = 1

parlay.items.parlay_standard module

class parlay.items.parlay_standard.**ParlayCommandItem**(*item_id=None*, *name=None*, *reactor=None*, *adapter=None*, *parents=None*)

Bases: *parlay.items.parlay_standard.ParlayStandardItem*

This is a Parlay Item that defines functions that serve as commands, with arguments.

This class enables you to use the `parlay_command()` decorator over your command functions. Then, those functions will be available as commands that can be called from the user interface, scripts, or by other items.

Example: How to define a class as a ParlayCommandItem:

```
from parlay import local_item, ParlayCommandItem, parlay_command

@local_item()
class MotorSimulator(ParlayCommandItem):

    def __init__(self, item_id, item_name):
        self.coord = 0
        ParlayCommandItem.__init__(self, item_id, item_name)

    @parlay_command
    def move_to_coordinate(self, coordinate)
        self.coord = coordinate
```

Example: How to instantiate an item from the above definition:

```
import parlay
from motor_sim import MotorSimulator

MotorSimulator("motor1", "motor 1")  # motor1 will be discoverable
parlay.start()
```

Example: How to interact with the instantiated item from a Parlay script:

```
# script_move_motor.py

setup()
discover()
motor_sim = get_item_by_name("motor 1")
motor_sim.move_to_coordinate(500)
```

SUBSYSTEM_ID = 'python'

get_discovery()
Will auto-populate the UI with inputs for commands

on_message(*msg*)
Will handle command messages automatically. Returns True if the message was handled, False otherwise

send_response(*msg*, *contents=None*, *msg_status='OK'*)

wait_for_next_recv_msg()
Returns a deferred that will callback on the next message we RECEIVE

wait_for_next_sent_msg()
Returns a deferred that will callback on the next message we SEND

class parlay.items.parlay_standard.**ParlayDatastream**(**args*, ***kwargs*)
Bases: *parlay.items.parlay_standard.ParlayProperty*

Deprecated

units = None

class parlay.items.parlay_standard.**ParlayProperty**(*default=None*, *val_type=<type 'str'>*, *read_only=False*, *write_only=False*, *custom_read=None*, *custom_write=None*, *callback=<function <lambda>>*)
Bases: object

A convenience class for creating properties of ParlayCommandItems.

Example: How to define a property:

```
class MyItem(ParlayCommandItem):
    x = ParlayProperty(default=0, val_type=int)

    def __init__(self, item_id, item_name):
        ParlayCommandItem.__init__(self, item_id, item_name)
    ...
```

Example: How to access a property from a script:

```
setup()
discover()
my_item = get_item_by_name("MyItem")
original_value = my_item.x
my_item.x = 5
```

listen(*instance*, *listener*, *requester_id*)
Listen to the datastream. Will call calback whenever there is a change

stop(*instance*, *requester_id*)
Stop listening

class parlay.items.parlay_standard.**ParlayStandardItem**(*item_id*, *name*, *reactor=None*, *adapter=None*, *parents=None*)
Bases: *parlay.items.threaded_item.ThreadedItem*

This is a parlay standard item. It supports building inputs for the UI in an intuitive manner during discovery. Inherit from it and use the parlay decorators to get UI functionality

add_datastream(*id*, *attr_name=None*, *units=''*, *name=None*)
Add a datastream to this Item. :param id : The id of the stream :param name: The name of the datastream (defaults to id) :param attr_name: the name of the attr to set in 'self' when setting and getting (same as name if None) :param units: units of streaming value that will be reported during discovery

add_field(*msg_key*, *input*, *label=None*, *required=False*, *hidden=False*, *default=None*, *dropdown_options=None*, *dropdown_sub_fields=None*, *topic_field=False*)
Add a field to this items discovery. This field will show up in the item's CARD in the UI

add_property(*id*, *attr_name=None*, *input='STRING'*, *read_only=False*, *write_only=False*, *name=None*)
Add a property to this Item. :param id : the id of the name :param name : The name of the property (defaults to ID) :param attr_name = the name of the attr to set in 'self' when setting and getting (None if same as name) :param read_only = Read only :param write_only = write_only

clear_fields()
clears all fields. Useful to change the discovery UI

create_field(*msg_key*, *input*, *label=None*, *required=False*, *hidden=False*, *default=None*, *dropdown_options=None*, *dropdown_sub_fields=None*)
Create a field for the UI

encode_binary_data(*mime_type*, *content*)
Encodes binary data into a mime format that can be displayed on the Parlay UI. :param mime_type: The mime type for the browser to understand the encoded data. :param content: The content to encode. :return: The encoded data with mime formatting.

get_discovery()
Discovery method. You can override this in a subclass if you want, but it will probably be easier to use the self.add_property and self.add_field helper methods and call this method like: def get_discovery(self):

discovery = ParlayStandardItem.get_discovery(self) # do other stuff for subclass here return discovery

send_event(*info*, *event*, *description*, *to=''*)
Broadcasts an event if no arguments for 'to' are supplied, direct event otherwise. :param info: Information about what caused the event. :param event: The event name or number to fire. :param description: The description of the event. :param to: If supplied, sends the events directly to the specified items instead of broadcasting. :type to: list[str] :return: None.

send_file(*filename*, *receiver=None*)
send file contents as an event message (EVENT is ParlaySendFileEvent) to a receiver

Parameters

- **filename** (*str*) – path to file that needs to be sent
- **receiver** (*str*) – ID that the file needs to be sent to. by default, the receiver is None meaning the file sending event should be broadcast. If not broadcast this will generally be "UI"

the item will send an event message. The contents will be formatted in the following way:

contents: { "EVENT": "ParlaySendFileEvent" "DESCRIPTION": [filename being sent as string], "INFO": [contents of file as string]

}

send_message(*to=None*, *from_=None*, *contents=None*, *tx_type='DIRECT'*, *msg_type='DATA'*, *msg_id=None*, *msg_status='OK'*, *response_req=False*, *extra_topics=None*)
Sends a Parlay standard message. contents is a dictionary of contents to send

send_parlay_command(*to*, *command*, ***kwargs*)
Send a parlay command to an known ID

parlay.items.parlay_standard.**parlay_command**(*async=False*, *auto_type_cast=True*)
Make the decorated method a parlay_command.

Parameters

- **async** – If True, will run as a normal twisted async function call. If False, parlay will spawn a separate thread and run the function synchronously (Default false)
- **auto_type_cast** – If true, will search the function's docstring for type info about the arguments, and provide that information during discovery

parlay.items.parlay_standard_proxys module

exception parlay.items.parlay_standard_proxys.**BadStatusError**(*error*, *description=''*)
Bases: exceptions.Exception

Throw this if you want to return a Bad Status!

class parlay.items.parlay_standard_proxys.**CommandHandle**(*msg*, *script*)
Bases: object

This is a command handle that wraps a command message and allows blocking until certain messages are recieved

wait_for(***kwargs*)
Block and wait for a message in our queue where fn returns true. Return that message

Parameters

- **fn** – function with argument message (parlay dictionary) that examines message and returns true if it meets certain criteria
- **timeout_sec** – if matching message not received within this time, raises parlay.utils.TimeoutError.

Returns message as Parlay dictionary

wait_for_ack(***kwargs*)
Waits until a response to the command is received with a message status of PROGRESS.

Parameters **timeout_sec** – if PROGRESS response not received within this time, raises parlay.utils.TimeoutError.

Returns message as Parlay dictionary

wait_for_complete(***kwargs*)
Waits until a response to the command is received, with a message status of OK or ERROR.

Parameters **timeout_sec** – if response not received within this time, raises parlay.utils.TimeoutError.

Returns the RESULT field of the response's CONTENTS if it exists, else all of CONTENTS

class parlay.items.parlay_standard_proxys.**ParlayStandardScriptProxy**(*discovery*, *script*)

Bases: object

A proxy class for the script to use, that will auto-detect discovery information and allow script writers to intuitively use the item

class PropertyProxy(*id*, *item_proxy*, *blocking_set=True*)

Bases: object

Proxy class for a parlay property

class StreamProxy(*id*, *item_proxy*, *rate*)

Bases: object

Proxy class for a parlay stream

MAX_LOG_SIZE = 1000000

attach_listener(*listener*)

clear_log()

Resets the internal log

get()

get_log()

Public interface to get the stream log

start_logging(*rate*)

Script function that enables logging. When a new value is pushed to the datastream the value will get pushed to the end of the log.

stop()

Stop streaming :return:

stop_logging()

Script function that disables logging.

wait_for_value()

If in thread: Will block until datastream is updated
If in Broker: Will return deferred that is called back with the datastream value when updated

get_datastream_handle(*name*)

send_parlay_command(*command*, ***kwargs*)

Manually send a parlay command. Returns a handle that can be paused on

parlay.items.threaded_item module

exception parlay.items.threaded_item.**AsyncSystemError**(*error_msg*)

Bases: exceptions.Exception

This error class is for asynchronous system errors.

exception parlay.items.threaded_item.**ErrorResponse**(*error_msg*)

Bases: exceptions.Exception

class parlay.items.threaded_item.**ListenerStatus**

Bases: object

Enum object for keeping or removing listeners

KEEP_LISTENER = False

REMOVE_LISTENER = True

class parlay.items.threaded_item.**ThreadedItem**(*item_id*, *name*, *reactor=None*, *adapter=None*, *parents=None*)

Bases: *parlay.items.base.BaseItem*

Base object for all Parlay scripts

add_listener(*listener_function*)
Add functions to the listener list

close_protocol(*protocol_id*)
close a protocol by id :param protocol_id: :return:

discover(*force=True*)
Run a discovery so that the script knows what items are attached and can get handles to them.

Parameters **force** – If True, will force a rediscovery, if False will take the last cached discovery

discovery = None
The current discovery information to pull from

get_all_items_with_name(***kwargs*)
Returns a handler object that can be used to send messages to an item.

Parameters **item_name** – globally unique name of the item

Returns a proxy object for the item

get_item_by_id(***kwargs*)
Returns a handler object that can be used to send messages to an item.

Parameters **item_id** – globally unique id of the item

Returns a proxy object for the item

get_item_by_name(***kwargs*)
Returns a handler object that can be used to send messages to an item.

Parameters **item_name** – globally unique name of the item

Returns a proxy object for the item

load_discovery(*path*)
Load discovery from a file.

Parameters **path** – The path to the file that has the JSON discovery

make_msg(*to*, *command*, *msg_type='COMMAND'*, *direct=True*, *response_req=True*, *_extra_topics=None*, ***kwargs*)
Prepare a message for the broker to disperse

open(*protocol*, ***params*)

Parameters

- **protocol** – protocol being used
- **params** – other parameters

Returns

publish(*msg*)

save_discovery(*path*)
Save the current discovery information to a file so it can be loaded later

Parameters **path** – The Path to the file to save to (Warning: will be overwritten)

send_parlay_message(*msg*, *timeout=600*, *wait=None*)
Send a command. This will be sent from the reactor thread. If a response is required, we will wait for it. :param msg The Message to send :param timeout If we require a response and don't get one back int timeout seconds, raise a timeout exception :param wait If set to True, will block until a response, if false will continue without blocking, If set to None, till auto discover based on message RESPONSE_REQ.

sleep(*timeout*)
Sleep for <timeout> seconds. This call is BLOCKING.

Parameters **timeout** – number of seconds to sleep

stop_reactor_on_close = True

parlay.items.threaded_item.**cleanup**()

Module contents

parlay.protocols package

Subpackages

parlay.protocols.pcom package

Submodules

parlay.protocols.pcom.enums module

parlay.protocols.pcom.pcom_message module

PCOM_Message.py

This is a message class that represents a middle ground between the high level JSON message and low level serial message.

Variables in this class will serve as storage points for the information inside of each message. The variables are accessed using properties (@property and @setter decorators).

There are two key functions in this class (from_json_msg() and to_json_msg()) which handle the conversion to and from a JSON message.

class parlay.protocols.pcom.pcom_message.**PCOMMessage**(*to=None*, *from_=None*, *msg_id=0*, *tx_type=None*, *msg_type=None*, *attributes=0*, *response_code=None*, *response_req=None*, *msg_status=None*, *contents=None*, *data=None*, *data_fmt=None*, *topics=None*, *description=''*)

Bases: object

GLOBAL_ERROR_CODE_ID = 255

VALID_JSON_MESSAGE_TYPES = ['COMMAND', 'EVENT', 'RESPONSE', 'PROPERTY', 'STREAM']

attributes

category()

data

classmethod from_json_msg(*json_msg*)
Converts a dictionary message to a PCOM message object

Parameters **json_msg** – JSON message

Returns PCOM message object

static get_name_from_id(*item_id*, *map*, *id_to_find*, *default_val='No name found'*)
Gets name from item ID. Assuming name is the KEY and ID is the value in <map> dictionary

Parameters

- **item_id** –
- **map** –
- **default_val** –

Returns

static get_subsystem(*id*)
" Gets the subsystem of the message.

get_tx_type_from_id(*id*)
Given an ID, returns the msg['TOPICS']['TX_TYPE'] that should be assigned :param id: destination item ID :return:

option()

sub_type()

to_json_msg()
Converts from PCOMMessage type to Parlay JSON message. Returns message when translation is complete. :return: Parlay JSON message equivalent of this object

parlay.protocols.pcom.pcom_serial module

pcom_serial.py

This protocol enables Parlay to interact with embedded devices. This class handles the passing of messages between Parlay and embedded devices.

class parlay.protocols.pcom.pcom_serial.**ACKInfo**(*sequence_number*, *num_retries*, *packet*, *failure_function*, *msg_deferred*)
Stores ACK information: deferred and number of retries

class parlay.protocols.pcom.pcom_serial.**PCOMSerial**(*adapter*, *port*)
Bases: *parlay.protocols.base_protocol.BaseProtocol*, twisted.protocols.basic.LineReceiver

ACK_TIMEOUT = 0.25

BAUD_RATE = 115200

BROADCAST_SUBSYSTEM_ID = 32768

DISCOVERY_CODE = 65278

DISCOVERY_TIMEOUT_ID = 4278124544

EMBD_REACTOR_ID = 0

ERROR_STATUS = 4278059008

FTDI_VENDOR_ID = '=0403'

INVALID_SUBSYSTEM_ID = 255

ITEM_ID_MASK = 255

MESSAGE_TIMEOUT_ERROR_ID = 4278190080

MIN_EVENT_ID = 0

NUM_EVENT_ID_BITS = 16

NUM_RETRIES = 3

PCOM_RESET_ID = 65278

SEQ_BITS = 4

STLINK_STRING = 'Link'

STM_VCP_STRING = 'STM32 Virtual ComPort'

ST_SNR = 'SNR='

ST_VENDOR_ID = '=0483'

SUBSYSTEM_ID_MASK = 65280

SUBSYSTEM_SHIFT = 8

USB_SERIAL_CONV_STRING = 'USB Serial Converter'

WINDOW_SIZE = 8

ack_timeout_handler()
Called when an ACK fails to send :return: None

add_message_to_queue(*message*)
This will send a packet down the serial line. Subscribe to messages using the adapter

:param message : A parlay dictionary message

broadcast_error_message(*error_id*, *description*, *info*)
Broadcasts an error message to the broker. This is mainly used to signal a failed discovery.

Parameters

- **error_id** – ID of the error message
- **description** – description that will be placed under contents
- **info** – info that will be placed under contents

Returns None

static build_command_info(*format*, *input_params*, *output_params*)
Builds the command info (dictionary) for a command when given the format, input parameters, and output parameters.

Parameters

- **format** – format string (eg. "fff")

- **input_params** – list of parameter names (eg. [“rate”, “height”, “weight”])
- **output_params** – list of output parameter names (eg. [“Rick”, “and”, “Morty”])

Returns the command info dictionary

static build_parlay_item_subfields(*c_input_format*, *c_input_names*, *local_subfields*, *parlay_item*)

Builds up Parlay items to be pushed to discoverer

Parameters

- **c_input_format** – format string Eg. “bBIq”
- **c_input_names** – list of string names representing the input parameters Eg. [“hello”, “hello”]
- **local_subfields** – subfields of the Parlay item
- **parlay_item** – the Parlay item to be built up

Returns

static build_property_data(*name*, *fmt*)

Builds property dictionary consisting of the name and format of the property.

Parameters

- **name** – property name (eg. “Butterbot”)
- **fmt** – format string for the property (eg. “f”)

Returns dictionary representing information about the property

close()

Simply close the connection :return:

static command_cb(*command_info_list*, *item_id*, *command_id*, *command_dropdowns*, *command_subfields*, *parlay_item*, *hidden=False*)

Callback function used to update the command map and parlay item dropdown menu when the command info is retrieved from the embedded device during discovery. This function is called using gatherResults which will only callback once all deferreds have been fired.

Parameters **command_info_list** – Stores the command name, command input format, command input names

and command output description. This information will be used to populate the ParlayStandardItem. :param item_id: 2 byte item ID of the ParlayStandardItem we will be populating :param command_id: 2 byte command ID of the command we have the information for :param command_dropdowns: dropdown field for the ParlayStandardItem :param command_subfields: subfields for each dropdown option of the ParlayStandardItem :param parlay_item: ParlayStandardItem that we will be updating :param hidden: whether or not the command will be hidden from UI :return:

connectionLost(*reason=<twisted.python.failure.Failure twisted.internet.error.ConnectionDone: Connection was closed cleanly.>*)

Overridden function that is called when the connected port is disconnected. Simply sets the self.is_port_attached flag to False so that we know we have been disconnected from the target embedded board. :param reason: Reason for disconnection. Of Twisted failure type. :return:

connectionMade()

The initializer for the protocol. This function is called when a connection to the server (broker in our case) has been established.

get_command_input_param_format(***kwargs*)

Given a command ID and item ID, sends a message to the item ID requesting the format of its input

parameters. This functions should return a string that describes each parameter. NOTE: variable arrays are indicated with a *. Eg. A list of ints would be "*i". See format string details for character->byte translation. :param to: destination item ID :param requested_command_id: command ID that we want the parameter format of :return: format string describing input parameters

get_command_input_param_names(***kwargs*)
Given an item ID and a command ID, requests the parameter names of the command from the item. Returns a list of names (comma delimited) that represent the parameter names.

TODO: change return value to string?

Eg. "frequency,duty cycle" :param to: destination item ID :param requested_command_id: command id to find the parameter names of :return: a list of parameter names

get_command_name(***kwargs*)
Sends a messge down the serial line requesting the property name of a given property ID, used in discovery protocol :param to: destination ID :param requested_command_id: command ID that we want to know the name of :return: name from Embedded Core

get_command_output_parameter_desc(***kwargs*)
Given an item ID and a command ID, requests the output description Returns a list of names (comma delimited) that represent the output names

TODO: change return value to string?

Eg. "frequency,duty cycle" :param to: destination item ID :param requested_command_id: command id to find the parameter names of :return: a list of parameter names

get_discovery()
Hitting the "discovery" button on the UI triggers this generator.

Run a discovery for everything connected to this protocol and return a list of of all connected: items, messages, and endpoint types

classmethod get_open_params_defaults()
Returns a list of parameters defaults. These will be displayed in the UI. :return: default args: the default arguments provided to the user in the UI

get_property_desc(***kwargs*)
Sends a message to the embedded board requesting the property description for a specified property ID

Parameters

- **to** – item ID to send the message to
- **requested_property_id** – property ID to get the description of

Returns

get_property_name(***kwargs*)
Sends a message down the serial line requesting the command name of a given command ID, used in discovery protocol :param to: destination item ID :param requested_property_id: property ID that we want to know the name of :return: name of the property from Embedded Core

get_property_type(***kwargs*)
Given a property ID, requests the property's type from the item ID. Gets back a format string.

Parameters

- **to** – destination item ID
- **requested_property_id** – property ID that we want the type of

Returns format string describing the type

static initialize_command_maps(*item_id*)
Creates the discovery command entries in the command map for the specified item ID. :param item_id: Item ID found during discovery. :return: None

lineReceived(*line*)
If this function is called we have received a <line> on the serial port that ended in 0x03.

Parameters **line** –

Returns

load_discovery_from_file()
Loads discovery info from PCOMSerial.discovery_file that was passed in at protocol open.

Returns discovery message that should be sent to broker

msg_timeout_errback(*failure*)
Errback attached to a message that is called if it fails to send.

Parameters **failure** –

Returns defer.failure.Failure object

classmethod open(*adapter*, *port=None*, *discovery_file=None*, *low_speed=False*)

Parameters

- **cls** – The class object
- **adapter** – current adapter instance used to interface with broker
- **port** – the serial port device to use. Leave this parameter empty if autoconnect is desired.
- **low_speed** – limits discovery messaging to one at a time for slower devices

Returns returns the instantiated protocol object

‘

process_data_file(*data*)
Given the data from the discovery file, fills in the corresponding maps. :param data: data produces from JSON file :return: discovery message to produce

static property_cb(*property_info_list*, *item_id*, *property_id*, *parlay_item*)
Callback function that populates the ParlayStandardItem parlay_item with the designated property information.

Parameters

- **property_info_list** – Property name and property type obtained from the embedded device
- **item_id** – 2 byte item ID that we will be populating the ParlayStandardItem of
- **property_id** – 2 byte property ID that represents the property we will updating
- **parlay_item** – ParlayStandardItem object

Returns

rawDataReceived(*data*)
This function is called whenever data appears on the serial port and raw mode is turned on. Since this protocol uses line receiving, this function should never be called, so raise an excpetion if it is.

Parameters **data** –

Returns

reset()
Resets the information relevant to layer 2 communications (event ID generator, ACK window). :return: None

send_command(*to=None*, *tx_type='DIRECT'*, *command_id=0*, *msg_status='INFO'*, *response_req=True*, *params=[]*, *data=[]*)
Send a command and return a deferred that will succeed on a response and with the response

Parameters

- **to** – destination item ID
- **tx_type** – DIRECT or BROADCAST
- **command_id** – ID of the command
- **msg_status** – status of the message: ERROR, WARNING, INFO, PROGRESS, or OK
- **response_req** – boolean whether or not a response is required
- **params** – command parameters
- **data** – data that corresponds to each parameter

Returns

send_error_message(*original_message*, *message_status*, *description=''*)
Sends a notification error to the destination ID.

Parameters

- **original_message** – PCOM Message object that holds the IDs of the sender and receiver
- **message_status** – Message status code that translates to an error message.
- **description** – description for the error message to be thrown in CONTENTS

Returns

static tokenize_format_char_string(*format_chars*)
Given a format string, returns a list of tokens.

Eg.

"bB*i" -> ["b", "B", "*i"] :param format_chars: :return:

write_discovery_info_to_file(*file_name*, *discovery_msg*)
Given a discovery message and file name, writes the necessary discovery information to the file.

This includes several maps and the discovery message itself so that it does not need to be generated.

Parameters

- **file_name** – discovery file name
- **discovery_msg** – discovery message to be pushed to broker

Returns

write_to_port(*buf*)
Helper function used to write to the COM port. If a valid port is not attached it attempts to reopen it first. :param buf: Buffer that is going to be written to the COM port :return: None

class parlay.protocols.pcom.pcom_serial.**SlidingACKWindow**(*window_size*, *num_retries*, *_protocol*)

Represents an ACK window

EXPIRED = 1001

TIMEOUT = 1000

ack_received_callback(*sequence_number*)

Callback for the deferred objects in the sliding ACK window.

When an ACK is received we should remove it from the window and then move one ACK from the queue into the window :return:

ack_timeout(*d*, *seconds*, *sequence_number*)

An extension of the timeout() function from Parlay utils. Calls the errback of d in <seconds> seconds if d is not called. In this case we will be passing a TimeoutException with the ACK sequence number so that we can remove it from the table.

ack_timeout_errback(*timeout_failure*)

Errback that is called on ACK timeout :param timeout_exception: TimeoutException object that holds the ACK sequence number that timed out :return:

add(*ack_info*)

Adds ack_info to the window if there is room, or to the queue if there isn't any room :param ack_info: ACKInfo object :return:

add_to_window(*ack_info*)

Adds <ack_info> to the window :param ack_info: ACKInfo object that will be added :return:

remove(*sequence_number*)

Removes ack_info from the window :param sequence_number: sequence number of the ACK to remove from window :return:

reset_window()

exception parlay.protocols.pcom.pcom_serial.**TimeoutException**(*sequence_number*)

Bases: exceptions.Exception

A custom exception used to be passed to the timeout errback for ACKs. The sequence number needs to be stored so that the errback can lookup the correct ACK in the sliding window.

parlay.protocols.pcom.serial_encoding module

Serial_encoding.py

A collection of helper functions that aid in the packing and unpacking of binary data as part of the PCOM Serial Protocol.

exception parlay.protocols.pcom.serial_encoding.**FailCRC**

Bases: exceptions.Exception

exception parlay.protocols.pcom.serial_encoding.**InvalidPacket**

Bases: exceptions.Exception

parlay.protocols.pcom.serial_encoding.**ack_nak_message**(*sequence_num*, *is_ack*)

Generate an Ack message with the packets sequence number

ACK_MASK | Seq num, CHECKSUM, PAYLOAD SIZE (2 bytes)|

parlay.protocols.pcom.serial_encoding.**cast_data**(*fmt_string*, *data*)

Returns a list of data casted according to the format string to prepare for packing.

When using the struct library the data that is going to be packed must match the type of the format string character that it maps to.

Eg.

struct.pack("?", True)

requires a boolean. The data received from the JSON message is a string, so we need to use this funtion to map it to the correct type.

Example usage:

cast_data("*B", ["12, 13, 14"]) –> [[12, 13, 14]] cast_data("Hs", ["12", "test"] –> [12, "test"]

The return list will be sent to struct.pack()

Parameters

- **fmt_string** – String of format characters from format string table
- **data** – List of strings representing the data that will be sent down

the serial line :return: List of data that is now casted to the correct type according to the format string

parlay.protocols.pcom.serial_encoding.**convert_to_bool**(*bool_obj*)

Helper function to convert strings to boolean for casting purposes :param bool_string: boolean in string format, eg. "False" , or "True" :return: bool value

parlay.protocols.pcom.serial_encoding.**decode_pcom_message**(*binary_msg*)

Build a pcom message object from the serialized message :type binary_msg: str :return : PCOMMessage

parlay.protocols.pcom.serial_encoding.**encode_pcom_message**(*msg*)

Build the base binary message without the data sections :type msg: PCOMMessage

Bytes [0:1] Event ID (Unique ID of event) Bytes [2:3] Source ID Bytes [4:5] Destination ID Bytes [6:7] Order/response code (Command ID, property ID, or status code depending on event type) Bytes [8:9] Message Status Bytes [10] Type (Type and subtype of event) Bytes [11] Attributes (Event attributes) Bytes [12:N] Format string (Null terminated data structure description (0 for no data)) Bytes [N+1:M] Data in the form of bytes[10:N]. Size must match format string

format_string: Describes the structure of the data using a character for each type.

Type | Character | # bytes |

|———————-|———————|———————| | unsigned byte | B | 1 | |———————-|———————|———————| | signed byte | b | 1 | |———————-|———————|———————| | padding | x | 1 | |———————-|———————|———————| | character | c | 1 | |———————-|———————|———————| | unsigned short | H | 2 | |———————-|———————|———————| | signed short | h | 2 | |———————-|———————|———————| | unsigned int | I | 4 | |———————-|———————|———————| | signed int | i | 4 | |———————-|———————|———————| | unsigned long | Q | 8 | |———————-|———————|———————| | signed long | q | 8 | |———————-|———————|———————| | float | f | 4 | |———————-|———————|———————| | double | d | 8 | |———————-|———————|———————| | string | s | ? | |———————-|———————|———————| | bool | ? | 1 | |———————-|———————|———————|

parlay.protocols.pcom.serial_encoding.**expand_fmt_string**(*format_string*)

Expands the format string to use preexisting logic to cast the data accordingly.

Example usage:

expand_fmt_string("3H") –> "HHH"

Parameters **format_string** – string of format chars from the format string table

Returns format string that has been expanded from the condensed form.

parlay.protocols.pcom.serial_encoding.**flatten**(*data*)
Flattens an irregular data list into one list.

Eg.

[1, 2, [3, 4]] -> [1, 2, 3, 4]

NOTE: used for handling variable length format string data

FORMAT STRING = b*b DATA = [2, [3, 4]]

Data needs to be packed as struct.pack('b2b', 2, 3, 4)

Parameters **data** – data list to be packed

Returns flattened data lilst

parlay.protocols.pcom.serial_encoding.**get_category**(*message*)
Gets the serial category of the PCOM Message.. 0 – Order 1 – Order Response 2 – Notification

Parameters **message** – PCOM Message that we want the category of

Returns Category of the PCOM message, will be a number between 0-2.

parlay.protocols.pcom.serial_encoding.**get_checksum**(*packet_sum*)
Calculates the checksum given the summation of the packet :param packet_sum: sum of all bytes in the packet :return: checksum for the packet

parlay.protocols.pcom.serial_encoding.**get_option**(*msg*, *cat*, *sub_type*)
Returns the option portion of the message type.

When serializing the message type (1 byte) the format follows:

bytes[6:7] – category bytes[4:5] – sub type bytes[0:3] – option

Given a PCOM Message, and the message's sub type and category this function will return the corresponding option so that it may be serialized

Parameters

- **msg** – PCOM Message we will be getting the option of
- **cat** – category of the PCOM message
- **sub_type** – sub type of the PCOM message

Returns the option of the message (will be a number between 0-4)

parlay.protocols.pcom.serial_encoding.**get_str_len**(*bin_data*)
Helper function that gets the length of a binary sequence up to and including the NULL byte.

Parameters **bin_data** – binary byte sequence

Returns count (including NULL byte) of bytes

parlay.protocols.pcom.serial_encoding.**get_sub_type**(*msg*, *category*)
Extracts the sub type of the PCOM message passed as the parameter msg

Parameters

- **msg** – PCOM message that we will be getting the sub type of
- **category** – category of the PCOM message that we want the subtype of

Returns the subtype of the message (will be a number between 0-4)

parlay.protocols.pcom.serial_encoding.**hex_print**(*buf*)
Prints the characters of buffer in hexidecimal :param buf: buffer to be printed :return:

parlay.protocols.pcom.serial_encoding.**p_wrap**(*stream*)
Do the promenade wrap! The promenade protocol looks like:

START BYTE <byte sequence> END BYTE

Where START BYTE and END BYTE are 0x02 and 0x02 (for now at least).

Since there is a possibility of 0x02 and 0x03 appearing in the data stream we must added 0x10 to all 0x10, 0x02, 0x03 bytes and 0x10 should be inserted before each "problem" byte.

For example

stream = 0x03 0x04 0x05 0x06 0x07 p_wrap(stream) = 0x02 0x10 0x13 0x04 0x05 0x06 0x07 0x03

Parameters **stream** – A raw stream of bytes

Returns A bytearray that has been run through the Promenade protocol

parlay.protocols.pcom.serial_encoding.**pack_little_endian**(*type_string*, *data_list*)

Parameters

- **type_string** –
- **data_list** –

Returns

parlay.protocols.pcom.serial_encoding.**serialize_msg_attrs**(*msg*)

Parameters **msg** – A Message object that was translated from a Parlay JSON message

Returns A byte representing the attributes field of the byte sequence. This

will be sent to the embedded core.

parlay.protocols.pcom.serial_encoding.**serialize_msg_type**(*msg*)
Converts the message type to a binary sequence. :param msg: PCOM Message :return:

parlay.protocols.pcom.serial_encoding.**serialize_response_code**(*message*)

Parameters **message** – PCOM message object that we will be serializing

Returns

parlay.protocols.pcom.serial_encoding.**sum_packet**(*msg*)
Calculate the checksum for the given msg

parlay.protocols.pcom.serial_encoding.**translate_fmt_str**(*fmt_str*, *data*)
Given a format string used in the Embedded Core Protocol and a binary message returns a format string that may be used in the Python struct library.

More specifically, the variable "s" format character is translated to "<str len>s".

eg:

translate_fmt_str("s", "ed") –> "3s" because the length of the string is 3 (including NULL byte).

Parameters

- **fmt_str** – A format string where 's' represents a variable length
- **data** – Binary sequence or list of bytes where strings are NULL terminated

Returns a new format string where 's' is replaced by '<len>s' where len is the length

of the string represented by 's'.

parlay.protocols.pcom.serial_encoding.**unstuff_packet**(*packet*)
Unstuff the packet. Descape and return sequence number, ack_expected, is_ack, and is_nak, dict_msg as a tuple dict_msg is the message (if there is one) or None (if it's an ack/nak)

parlay.protocols.pcom.serial_encoding.**wrap_packet**(*packet*, *sequence_num*, *use_ack*)
Appends sequence number + packet type, checksum, and payload length to the payload.

Also escapes packet according to the pcom protocol.

Parameters

- **packet** – packet that will be wrapped
- **sequence_num** – current sequence number (nibble)
- **use_ack** – whether an ack is required or not

Returns the resulting wrapped packet

Module contents

Submodules

parlay.protocols.base_protocol module

class parlay.protocols.base_protocol.**BaseProtocol**
Bases: object

This the base protocol that *all* parlay protocols must inherit from. Subclass this to make custom protocols to talk to 3rd party equipment.

close()
Override this with a generic method that will close the protocol.

e.g. def close():

get_discovery()
This will get called when a discovery message is sent out. Return a deferred that will be called back with all attached: item types, message types, and connected item instances

get_new_data_wait_handler()

classmethod get_open_params()
Returns the params for the cls.open() class method. Feel free to overwrite this in a sub-class if this default implementation doesn't fit your protocol's needs. :return: A list of parameter names :rtype: list

classmethod get_open_params_defaults()
return the defaults for parameters to the cls.open() using inspect. Feel free to overwrite this in a sub-class if this default implementation doesn't fit your protocol's needs. :return: A dictionary of parameter names -> default values. :rtype: dict

get_protocol_discovery_meta_info()
This will return protocol meta-info that will be returned with every discovery message. This is a good place to store things like enuemrations or protocol status to pass to the UI

got_new_data(***kwargs*)
Call this when you have new data and want to pass it to any waiting Items

classmethod open(*adapter*)
Override this with a generic method that will open the protocol. The first argument must be the broker, the rest will be parameters that the user can set. Default arguments will be sent to the UI and can be overwritten bu the user It must return the built protocol (subclass of Protocol) that will be registered with the Broker Be sure to decorate it with @classmethod

e.g. @classmethod def open(cls, adapter, ip, port=8080):

return protocol(ip,port)

class parlay.protocols.base_protocol.**WaitHandler**(*deferred*)
Bases: object

An Object used to do safe cross thread waits on deferreds

addCallback(***kwargs*)
Add a callback when you get new data

wait(***kwargs*)
Call this to wait until there is data from the protocol. If threaded: Will block. Return value is serial line data If Async : Will not block. Return value is Deferred that will be called back with data :param timeout_secs : Timeout if you don't get data in time. None if no timeout :type timeout_secs : int|None

parlay.protocols.local_item module

The Local Item protocol lets you open arbitrary items that have been registered as local

class parlay.protocols.local_item.**LocalItemProtocol**(*item*)
Bases: *parlay.protocols.base_protocol.BaseProtocol*

ID = 0

class TransportStub
Bases: object

Fake transport that will allow the protocol to think its writing to a transport

write(*payload*)

classmethod close()
Override this with a generic method that will close the protocol.

e.g. def close():

classmethod get_open_params_defaults()
return the defaults for parameters to the cls.open() using inspect. Feel free to overwrite this in a sub-class if this default implementation doesn't fit your protocol's needs. :return: A dictionary of parameter names -> default values. :rtype: dict

classmethod open(*broker*, *item_name*)
Override this with a generic method that will open the protocol. The first argument must be the broker, the rest will be parameters that the user can set. Default arguments will be sent to the UI and can be overwritten bu the user It must return the built protocol (subclass of Protocol) that will be registered with the Broker Be sure to decorate it with @classmethod

e.g. @classmethod def open(cls, adapter, ip, port=8080):

return protocol(ip,port)

classmethod open_for_obj(*item_obj*)

parlay.protocols.local_item.**auto_start**()
Auto start local items that have that flag set

parlay.protocols.local_item.**local_item**(*auto_connect=False*)
A class decorator for python Items that are not part of an external protocol.

Local items are self-contained, and do not communicate over external protocols, for example a serial port. They are typically used for simulators or pure python computation items.

Parameters **auto_connect** – whether to automatically connect to the Parlay broker when the item is created.

Returns decorator function

Example usage of local_item decorator:

```
# motor_sim.py

@local_item()
class MotorSimulator(ParlayCommandItem):

    def __init__(self, item_id, item_name):
        ...
```

Example usage of defined local item:

```
import parlay
from motor_sim import MotorSimulator

MotorSimulator("motor1", "motor 1")  # motor1 will be discoverable
parlay.start()
```

parlay.protocols.meta_protocol module

Define the base Protocol classes and meta-classes.

For documentation on broker and common message types see parlay.protocols:

```
.. py:exception:: InvalidProtocolDeclaration
```

module parlay.protocols.meta_protocol

Bases: exceptions.Exception

Raised when there was a problem with your protocol declaration

class parlay.protocols.meta_protocol.**ProtocolMeta**(*name*, *bases*, *dct*)
Bases: type

Meta-Class that will keep track of *all* message types declared Also builds the message field lookups from the Django-model-style message class definitions

protocol_registry = {'BaseProtocol': <class 'parlay.protocols.base_protocol.BaseProto

parlay.protocols.serial_line module

class parlay.protocols.serial_line.**ASCIILineProtocol**(*port*)
Bases: *parlay.protocols.base_protocol.BaseProtocol*, twisted.protocols.basic.LineReceiver

When a client connects over a serial, this is the protocol that will handle the communication. The messages are encoded as a JSON string

broker = <parlay.server.broker.Broker object>

close()
Override this with a generic method that will close the protocol.

e.g. def close():

classmethod get_open_params_defaults()
Override base class function to show dropdowns for defaults

lineReceived(*line*)
Override this for when each line is received.

@param line: The line which was received with the delimiter removed. @type line: C{bytes}

classmethod open(*broker*, *port='/dev/tty.usbserial-FTAJOUB2'*, *baudrate=57600*, *delimiter='\n'*, *bytesize=8*, *parity='N'*, *stopbits=1*)
This will be called bvy the system to construct and open a new SSCOM_Serial protocol :param cls : The class object (supplied by system) :param broker: current broker instance (supplied by system) :param port: the serial port device to use. On linux, something like "/dev/ttyUSB0". On windows something like "COM0" :param baudrate: baudrate of serial connection :param delimiter: The delimiter to token new lines off of. :param bytesize: The number of data bits. :param parity: The number of parity bits. :param stopbits: The number of stop bits.

open_ports = set([])

rawDataReceived(*data*)
Override this for when raw data is received.

class parlay.protocols.serial_line.**LineItem**(*item_id*, *name*, *protocol*)
Bases: *parlay.items.parlay_standard.ParlayCommandItem*

LAST_LINE_RECEIVED = ''

send_and_wait(***kwargs*)
Send and then wait for a single response

send_raw_data(***kwargs*)

wait_for_data(***kwargs*)
:type timeout_secs float

class parlay.protocols.serial_line.**USBASCIILineProtocol**(*port*)
Bases: *parlay.protocols.serial_line.ASCIILineProtocol*

This protocol should be used instead of ASCIILineProtocol when using a USB->Serial adapter to communicate serially with devices. Additional filtering options can be provided in the open() function in order to automatically connect and open the device.

DEFAULT_BAUD = 115200

NUM_REQUIRED_MATCHING_PORTS = 1

exception USBASCIIException(*msg*)
Bases: exceptions.Exception

Class used for generating exceptions from the USBASCIILineProtocol class.

ERROR_STRING_HEADER = '[USBASCIILineProtocol]'

classmethod get_open_params_defaults()
Used for determining default options for open() function call through Parlay ecosystem :return: Dictionary of the default options

classmethod open(*adapter*, *port_vendor_id=None*, *port_product_id=None*, *port_descriptor_regex_string=None*, *delimiter='\n'*, *baudrate=115200*, *bytesize=8*, *parity='N'*, *stopbits=1*)
Called by the Parlay ecosystem to construct a new USBASCIILineProtocol instance. For example, this function is called when the UI button "connect" is pressed. :param adapter: Current adapter instance. :param port_descriptor_regex_string: Regex string that will be matched with the USB descriptor. For example,

port_name_regex_string = "ST(|-)LINK"

can be used for matching "ST LINK" or "ST-LINK"

Parameters **port_vendor_id** – Vendor ID of the USB port we wish to automatically connect to. NOTE: Both vendor ID

and product ID must match the port. :param port_product_id: Product ID of the USB port we wish to automatically connect to. NOTE: Both vendor ID and product ID must match the port. :param delimiter: The delimiter to token new lines off of. :param baudrate: The baud rate of the serial connection. :param bytesize: Size of each byte for the serial connection. :param parity: Bit parity of the serial connection. :param stopbits: Number of stop bits for the serial connection. :return: Instantiated protocol

parlay.protocols.tcp module

class parlay.protocols.tcp.**TCPClientItem**(*item_id*, *item_name*, *protocol*)
Bases: *parlay.items.parlay_standard.ParlayCommandItem*

raw_data_received(*data*)
Override this method to deal with data incoming from the TCP stream. :param data: chunk of raw bytes received :return: None

send_raw_data(***kwargs*)
Send raw bytes over the TCP connection. :param data: raw bytes to send :type data: str :return: None

class parlay.protocols.tcp.**TCPClientProtocol**(*adapter*, *ip*, *port*)
Bases: *parlay.protocols.base_protocol.BaseProtocol*, twisted.internet.protocol.Protocol

close()
Override this with a generic method that will close the protocol.

e.g. def close():

connect(*adapter*, *ip*, *port*)
Establish a new TCP connection and link it with this protocol.

connect_failed(*failure*)

connectionLost(*reason=None*)
Called when the underlying TCP connection is lost.

connectionMade()
Called when the underlying TCP connection is made.

dataReceived(*data*)
Called with a chunk of raw bytes from the underlying TCP stream.

Override this method if you want to process raw bytes before sending them to the attached Parlay items. For example, you could buffer the raw data and look for a delimiting character to separate the raw bytes into messages.

Parameters **data** (*str*) – raw bytes received by the TCP stream

Returns None

classmethod get_open_params_defaults()
Gives the default arguments for the open() parameters. :return: The dictionary containing the default arguments.

classmethod open(*adapter*, *ip*, *port*)
Open a TCP Protocol. This does not actually make the TCP connection to the target IP address and port. That will be done when there is data to send.

Parameters

- **adapter** – Parlay adapter
- **ip** (*str*) – target IP address (example "192.168.0.2")
- **port** (*int*) – target TCP port

Returns the protocol object

send_raw_data(*data*)
Send raw bytes over the TCP connection. If the TCP connection is not currently open, open it, then send the data when it is open.

Note that even with an active connection, sending data may not happen immediately. The OS-level TCP stack may buffer data to consolidate into more efficient IP packets. If this is an issue, you may need to disable Nagle's algorithm, via *self.transport.setTcpNoDelay(True)*.

Parameters **data** (*str*) – raw bytes to send

Returns None

parlay.protocols.utils module

Generic utilities and helper functions that help make protocol development easier

class parlay.protocols.utils.**MessageQueue**(*callback*)
Bases: object

A basic message queue for messages that need to be acknowledged

add(*message*)
Add a message to the queue self.waiting_for_ack_seq_num = None # None = not waiting ... an int here, It will be sent in FIFO order

:returns A deferred that will be called back (or err-backed) when the message is done processing :rtype defer.deferred

class parlay.protocols.utils.**PrivateDeferred**(*canceller=None*)
Bases: twisted.internet.defer.Deferred

A Private Deferred is like a normal deferred, except that it can be passed around and callbacks can be attached by anone, but callback() and errback() have been overridden to throw an exception. the private _callback() and _errback() must be used

This ensures that only a user that 'knows what their doing' can issue the callback

callback(*result*)

errback(*fail=None*)

parlay.protocols.utils.**delay**(*seconds*)
Calls the function after a certain amount of time has passed. :param seconds: The amount of time to wait in seconds before calling. :return: A deferred.

parlay.protocols.utils.**message_id_generator**(*radix*, *minimum=0*)
makes an infinite iterator that will be modulo radix

parlay.protocols.utils.**timeout**(*d*, *seconds*)
Call d's errback if it hasn't been called back within 'seconds' number of seconds If 'seconds' is None, then do nothing

parlay.protocols.websocket module

class parlay.protocols.websocket.**WebSocketServerAdapter**(*broker=None*)
Bases: autobahn.twisted.websocket.WebSocketServerProtocol, *parlay.server.adapter.Adapter*

When a client connects over a websocket, this is the protocol that will handle the communication. The messages are encoded as a JSON string

broker = <parlay.server.broker.Broker object>

discover(*force*)
Return the discovery list (or a deferred) for all protocols and items attached to this adapter.

The final return value from this function (or its deferred) is a list of discovery dictionaries, as specified under the "Item Discovery" section of the Parlay Standard Item Communications Specification.

:type force bool :param force True if this requested discover action wants to clear any caches and do a fresh discover. :rtype defer.Deferred

get_open_protocols()

get_protocols()
Return a list of protocols that could potentially be opened. Return a deferred if this is not ready yet

onClose(*wasClean*, *code*, *reason*)
Implements autobahn.websocket.interfaces.IWebSocketChannel.onClose()

onConnect(*request*)
Callback fired during WebSocket opening handshake when new WebSocket client connection is about to be established.

When you want to accept the connection, return the accepted protocol from list of WebSocket (sub)protocols provided by client or *None* to speak no specific one or when the client protocol list was empty.

You may also return a pair of *(protocol, headers)* to send additional HTTP headers, with *headers* being a dictionary of key-values.

Throw autobahn.websocket.types.ConnectionDeny when you don't want to accept the WebSocket connection request.

Parameters **request** (instance of autobahn.websocket.protocol.ConnectionRequest) – WebSocket connection request information.

onMessage(*payload*, *isBinary*)
Implements autobahn.websocket.interfaces.IWebSocketChannel.onMessage()

send_message_as_JSON(*msg*)
Send a message dictionary as JSON

class parlay.protocols.websocket.**WebsocketClientAdapter**(*reactor*)
Bases: *parlay.server.adapter.Adapter*, autobahn.twisted.websocket.WebSocketClientProtocol

Connect a Python item to the Broker over a Websocket

call_on_every_message(*listener*)

onConnect(*request*)
Callback fired directly after WebSocket opening handshake when new WebSocket server connection was established.

Parameters **response** (instance of autobahn.websocket.protocol.ConnectionResponse) – WebSocket connection response information.

onMessage(*packet*, *isBinary*)
We got a message. See who wants to process it.

publish(*msg*, *callback=None*)

Parameters

- **msg** (*dict*) – Parlay message to publish
- **callback** (*function*) – Optional the callback function to call if the broker responds directly

Returns None

subscribe(*_fn=None*, ***topics*)
Subscribe to messages the topics in **kwargs

class parlay.protocols.websocket.**WebsocketClientAdapterFactory**(**args*, ***kwargs*)
Bases: autobahn.twisted.websocket.WebSocketClientFactory

buildProtocol(*addr*)
Create an instance of a subclass of Protocol.

The returned instance will handle input on an incoming server connection, and an attribute "factory" pointing to the creating factory.

Alternatively, L{None} may be returned to immediately close the new connection.

Override this method to alter how Protocol instances get created.

@param addr: an object implementing L{twisted.internet.interfaces.IAddress}

Module contents

parlay.server package

Submodules

parlay.server.adapter module

class parlay.server.adapter.**Adapter**
Bases: object

Adapters connect outside systems to the Broker. The Broker *only* connects with Adapters. Adapters handle opening protocols in their system, discovery of their system and relaying pub/sub messages to the broker.

Examples of supported adapters are: WebsocketServerAdapter, Pyadapter, FileTransportServerAdapter

DEFAULT_ADAPTER = <parlay.server.adapter.PyAdapter object>

deregister_item(*item*)
Register an item with the adapter :param item: item object to register :return: None

discover(*force*)
Return the discovery list (or a deferred) for all protocols and items attached to this adapter.

The final return value from this function (or its deferred) is a list of discovery dictionaries, as specified under the "Item Discovery" section of the Parlay Standard Item Communications Specification.

:type force bool :param force True if this requested discover action wants to clear any caches and do a fresh discover. :rtype defer.Deferred

get_open_protocols()

get_protocols()
Return a list of protocols that could potentially be opened. Return a deferred if this is not ready yet :rtype defer.Deferred

open_protocol(*protocol_name*, *protocol_args*)
open the protocol with the name protocol_name or raise a LookupError if there is no such protocol by that name :raise LookupError :arg protocol_name : The name of the protocol to open :arg protocol_args : A dict of key value pairs for the opening arguments :type protocol_name : str :type protocol_args : dict :rtype defer.Deferred

publish(*msg*, *callback=None*)

Parameters

- **msg** (*dict*) – Parlay message to publish
- **callback** (*function*) – Optional the callback function to call if the broker responds directly

Returns None

register_item(*item*)
Register an item with the adapter :param item: item object to register :return: None

subscribe(*fn*, ***kwargs*)
Subscribe to messages matching the provided topic keyword/value pairs :param fn: the listener function to call with messages meeting the criteria :type fn: function :param kwargs: The topics and their values to subscribe to :type kwargs: dict :return: None

class parlay.server.adapter.**PyAdapter**(*broker*)
Bases: *parlay.server.adapter.Adapter*

Adapter for the Python Broker and Python environment

deregister_item(*item*)
Register an item with the adapter :param item: item object to register :return: None

discover(*force*)
Return the discovery (or a deferred) for all protocols and items attached to this adapter :type force bool :param force True if this requested discover action wants to clear any caches and do a fresh discover.

get_open_protocols()
Returns a list of protocol object that are currently open :return:

get_protocols()
Return a list of protocols that could potentially be opened

open_protocol(*protocol_name*, *open_params*)
open the protocol with the name protocol_name or raise a LookupError if there is no such protocol by that name :raise LookupError :arg protocol_name : The name of the protocol to open :arg open_params : A dict of key value pairs for the opening arguments :type protocol_name : str :type open_params : dict

publish(*msg*, *callback=None*)

Parameters

- **msg** (*dict*) – Parlay message to publish
- **callback** (*function*) – Optional the callback function to call if the broker responds directly

Returns None

subscribe(*fn*, ***kwargs*)
Subscribe to messages matching the provided topic keyword/value pairs :param fn: the listener function to call with messages meeting the criteria :type fn: function :param kwargs: The topics and their values to subscribe to :type kwargs: dict :return: None

track_open_protocol(*protocol*)
track the given protocol for discovery

untrack_open_protocol(*protocol*)
Untracks the given protocol. You must call this when a protocol has closed to clean up after it.

parlay.server.advertiser module

class parlay.server.advertiser.**ParlayAdvertiser**
Bases: twisted.internet.protocol.DatagramProtocol

This protocol will advertise that a Parlay system is on this machine

datagramReceived(*datagram*, *address*)

startProtocol()
Called after protocol has started listening.

class parlay.server.advertiser.**ParlayConsumer**
Bases: twisted.internet.protocol.DatagramProtocol

This protocol will request for advertisements to be sent

datagramReceived(*datagram*, *address*)

print_output()

startProtocol()

parlay.server.advertiser.**main**()
Function to run if this is the entry point and not imported like a module :return:

parlay.server.broker module

class parlay.server.broker.**Broker**(*reactor*)
Bases: object

The Broker is the sole holder of global state. There should be only one. It also coordinates all communication between protocols.

class Modes
These are the modes that the broker can run in. * Development mode is purposefully easy to use an insecure to allow logging and easy control of the parlay system * Production mode is locked down and *more* secure (Security should always be validated independently)

DEVELOPMENT = 'DEVELOPER_MODE'

PRODUCTION = 'PRODUCTION_MODE'

classmethod call_on_start(*func*)
Call the supplied function when the broker starts OR if the broker has already started, call ASAP

classmethod call_on_stop(*func*)
Call the supplied function when the broker stops OR if the broker has already stopped, call ASAP

cleanup(*stop_reactor=True*)
called on exit to clean up the parlay environment

static get_instance()
@rtype Broker

static get_local_ip()

handle_broker_message(*msg*, *message_callback*)
Any message with topic type 'broker' should be passed into here. 'broker' messages are special messages that don't get 'published'. They are for querying the state of the system. 'broker' messages have a 'request' field and will reply with an appropriate 'response' field

message_callback is the function to call to send the message back to the protocol

handle_subscribe_message(*msg*, *message_callback*)

handle_unsubscribe_message(*msg*, *message_callback*)

instance = <parlay.server.broker.Broker object>

open_protocol(*protocol_name*, *open_params*)
Open a protocol with the given name and parameters (only run this once the Broker has started running

publish(*msg*, *write_method=None*)
Publish a message to the Parlay system :param msg : The message to publish :param write_method : the protocol's method to callback if the broker needs to send a response :type msg : dict

run(*mode='DEVELOPER_MODE'*, *ssl_only=False*, *use_ssl=False*, *open_browser=True*, *ui_path=None*, *ui_caching=False*)
Start up and run the broker. This method call with not return

static start(*mode='DEVELOPER_MODE'*, *ssl_only=False*, *open_browser=True*, *http_port=8080*, *https_port=8081*, *websocket_port=8085*, *secure_websocket_port=8086*, *ui_path=None*, *log_level=10*, *ui_caching=False*)
Run the default Broker implementation. This call will not return.

static start_for_test()

static stop()

static stop_for_test()

subscribe(*func*, *_owner_=None*, ***kwargs*)
Register a listener. The kwargs is a dictionary of args that **all** must be true to call this listener. You may register the same function multiple times with different kwargs, and it may be called multiple times for each message. @param func: The function to run @param kwargs: The key/value pairs to listen for

unsubscribe(*owner*, *TOPICS*)
Unsubscribe owner from all subscriptions that match TOPICS. Only EXACT matches will be unsubscribed

unsubscribe_all(*owner*, *root_list=None*)
Unsubscribe all function in our list that have a n owner that matches 'owner'

class parlay.server.broker.**BrokerSSlContextFactory**
Bases: twisted.internet.ssl.ContextFactory

A more secure context factory than the default one. Only supports high security encryption ciphers and exchange formats. Last Updated August 2015

getContext()
Return a SSL.Context object. override in subclasses.

parlay.server.broker.**main**()

parlay.server.broker.**run_in_broker**(*fn*)
Decorator: Wrap any method in this when you want to be sure it's called from the broker thread. If in a background thread, it will block until completion. If already in a reactor thread, then no change

parlay.server.broker.**run_in_thread**(*fn*)
Decorator: Wrap any method in this when you want to be sure it's called from a background thread . If in a background thread, no change. If in the broker thread, will move to background thread and return deferred with result.

parlay.server.http_server module

class parlay.server.http_server.**CacheControlledSite**(*ui_caching*, *resource*, *requestFactory=None*, **args*, ***kwargs*)
Bases: twisted.web.server.Site

Overloading twisted.server.Site to add HTTP headers for enabling/disabling browser cache

getResourceFor(*request*)

parlay.server.reactor module

We've added a number of enhancments to the Twisted reactor to enable easier threading handling. Instead of importing the twisted.internet reactor import this reactor like from parlay.server.reactor import reactor

class parlay.server.reactor.**ReactorWrapper**(*wrapped_reactor*)
Bases: object

in_reactor_thread()
Returns true if we're in the reactor thread context. False otherwise

maybeCallFromThread(*callable*, **args*, ***kwargs*)
If we're in a separate thread from the reactor, then call from the reactor thread. If we're in the reactor thread, then schedule for call later in 0 seconds

maybeDeferToThread(*callable*, **args*, ***kwargs*)
Call callable from DIFFERENT THREAD. If we're already in a different thread, then JUST CALL IT and return result If we're in the reactor thead, then call it in a different thread and return a deferred with the result

maybeblockingCallFromThread(*callable*, **args*, ***kwargs*)
Call callable from the reactor thread. If we are in the reactor thread, then call it and return a Deferred. If we are *not* in the reactor thread, then block on that deferred instal of returning it

run(*installSignalHandlers=True*)

parlay.server.reactor.**run_in_reactor**(*reactor*)
Decorator for automatically handling deferred <-> thread handoff. Any function wrapped in this will work in both a threaded context and a 'reactor' async context. Use this decorator if you want the function to always be run in the reactor context

:param reactor the Reactor to use (MUST BE A REACTORWRAPPER)

parlay.server.reactor.**run_in_thread**(*reactor*)
Decorator for automatically handling deferred <-> thread handoff. Any function wrapped in this will work in both a threaded context and a 'reactor' async context. Use this decorator if you want the function to always be run in the threaded context :param reactor the Reactor to use (MUST BE A REACTORWRAPPER)

parlay.server.serial_adapter module

class parlay.server.serial_adapter.**FileDeviceServerAdapter**(*filename*, *delimiter='n'*)
Bases: *parlay.server.serial_adapter.LineTransportServerAdapter*

This adapter is designed for communication ports that present themselves as regular device files, that CANNOT be opened as serial ports. Instead, we use the FileTransport transport layer and read/write them as regular files.

An example is the g_serial driver for Linux, whose device file cannot be opened by twisted.internet.serialport.SerialPort, but can be opened with the standard python open() function.

Example Usage:

```
from parlay import Broker, start

device_adapter = FileDeviceServerAdapter('/dev/ttyGS0')

broker = Broker.get_instance()
broker.adapters.append(device_adapter)
start()
```

class parlay.server.serial_adapter.**FileTransport**(*protocol*, *filename*, *reactor=None*, *mode='r+'*)
Bases: twisted.internet.abstract.FileDescriptor

Implements FileDescriptor abstract class with a simple file descriptor. Surprisingly, this is not included in Twisted.

Based heavily on twisted.internet.serialport.SerialPort

doRead()
Called when data is available for reading.

Subclasses must override this method. The result will be interpreted in the same way as a result of doWrite().

fileno()
File Descriptor number for select().

This method must be overridden or assigned in subclasses to indicate a valid file descriptor for the operating system.

flushInput()
Clear input buffer, discarding all that is in the buffer.

flushOutput()
Clear output buffer, aborting the current output and discarding all that is in the buffer.

writeSomeData(*data*)
Write as much as possible of the given data, immediately.

This is called to invoke the lower-level writing functionality, such as a socket's send() method, or a file's write(); this method returns an integer or an exception. If an integer, it is the number of bytes written (possibly zero); if an exception, it indicates the connection was lost.

class parlay.server.serial_adapter.**LineTransportServerAdapter**(*transport_factory*, *delimiter='n'*, ***kwargs*)
Bases: *parlay.server.adapter.Adapter*, twisted.protocols.basic.LineReceiver

Adapter class to connect the Parlay broker to a device (for example, serial) that implements the L{ITransport} interface.

DEFAULT_DISCOVERY_TIMEOUT_TIME = 3

broker = <parlay.server.broker.Broker object>

discover(*force*)
Sends a Parlay message of 'get_protocol_discovery' type via the transport. :param force: if False, return cached discovery if available. :type force: bool :return: Deferred to wait for discovery response

get_open_protocols()

get_protocols()
Return a list of protocols that could potentially be opened. Return a deferred if this is not ready yet :rtype defer.Deferred

lineReceived(*line*)
Handle a delimited line of bytes received by the transport. :param line: :return: None

send_message_as_json(*msg*)
Transforms parlay message dictionary to JSON, adds delimiting character, and sends it over the transport. :param msg: :return:

class parlay.server.serial_adapter.**SerialServerAdapter**(*port*, *baudrate=115200*, *delimiter='n'*)
Bases: *parlay.server.serial_adapter.LineTransportServerAdapter*

This class defines a Parlay server adapter for a serial port.

This class would be used when the Parlay broker will be running locally, and there is some device speaking the Parlay message protocol on the other side of a serial port. This adapter will allow that other device to connect to the Parlay broker.

Example Usage:

```
from parlay import Broker, start

serial_adapter = SerialServerAdapter('/dev/ttyUSB0', baudrate=57600)

broker = Broker.get_instance()
broker.adapters.append(serial_adapter)
start()
```

Module contents

parlay.test package

Submodules

parlay.test.integration_test_items module

parlay.test.test_base_item module

class parlay.test.test_base_item.**BaseItemTest**(*methodName='runTest'*)
Bases: twisted.trial._asynctest.TestCase, *parlay.testing.unittest_mixins.adapter.AdapterMixin*, *parlay.testing.unittest_mixins.reactor.ReactorMixin*

setUp()
Hook method for setting up the test fixture before exercising it.

testAttrError()

testNoChildren()

testNoParentDiscovery()

testSingleChildDiscovery()

testTwoChildrenDiscovery()

testTwoParentDiscovery()

class parlay.test.test_base_item.**NoSuperInitItem**(*item_id*, *item_name*)
Bases: *parlay.items.base.BaseItem*

parlay.test.test_items_parlay_standard module

class parlay.test.test_items_parlay_standard.**CommandTest**(*methodName='runTest'*)
Bases: twisted.trial._asynctest.TestCase, *parlay.testing.unittest_mixins.adapter.AdapterMixin*, *parlay.testing.unittest_mixins.reactor.ReactorMixin*

setUp()
Hook method for setting up the test fixture before exercising it.

testLocalAsyncCommand()

testLocalSyncCommand()

class parlay.test.test_items_parlay_standard.**CommandTestItem**(*item_id=None*, *name=None*, *reactor=None*, *adapter=None*, *parents=None*)
Bases: *parlay.items.parlay_standard.ParlayCommandItem*

Helper class to test custom commands

add(***kwargs*)

add_async(***kwargs*)

class parlay.test.test_items_parlay_standard.**PropertyTest**(*methodName='runTest'*)
Bases: twisted.trial._asynctest.TestCase, *parlay.testing.unittest_mixins.adapter.AdapterMixin*, *parlay.testing.unittest_mixins.reactor.ReactorMixin*

setUp()
Hook method for setting up the test fixture before exercising it.

tearDown()
Hook method for deconstructing the test fixture after testing it.

testCustomRWProp()

testDiscovery()

testPropertySpec_Get()

testPropertySpec_Set()

testPropertyTypeCoercion()

testSimpleProperty()

class parlay.test.test_items_parlay_standard.**PropertyTestItem**(*item_id=None*, *name=None*, *reactor=None*, *adapter=None*, *parents=None*)
Bases: *parlay.items.parlay_standard.ParlayCommandItem*

Helper class to test custom properties

custom_list = []

custom_rw_property = ''

read_only_property = None

simple_property = None

write_only_property = None

parlay.test.test_items_threaded_item module

class parlay.test.test_items_threaded_item.**ThreadedItemTest**(*methodName='runTest'*)
Bases: twisted.trial._asynctest.TestCase, *parlay.testing.unittest_mixins.adapter.AdapterMixin*, *parlay.testing.unittest_mixins.reactor.ReactorMixin*

setUp()
Hook method for setting up the test fixture before exercising it.

tearDown()
Hook method for deconstructing the test fixture after testing it.

testDiscovery()

testSleep()

parlay.test.test_protocols_base_protocol module

class parlay.test.test_protocols_base_protocol.**BaseProtocolTest**(*methodName='runTest'*)
Bases: twisted.trial._asynctest.TestCase, *parlay.testing.unittest_mixins.reactor.ReactorMixin*

setUp()
Hook method for setting up the test fixture before exercising it.

tearDown()
Hook method for deconstructing the test fixture after testing it.

testCrossThreadSuccess()

testSimpleWaitAsync()

testSimpleWaitSync()

testTimeoutSync()

parlay.test.test_server_broker module

class parlay.test.test_server_broker.**BrokerPubSubTests**(*methodName='runTest'*)
Bases: twisted.trial._asynctest.TestCase

setUp()
Hook method for setting up the test fixture before exercising it.

tearDown()
Hook method for deconstructing the test fixture after testing it.

testSimplePubSub()

testSimplePubSubNotPub()

class parlay.test.test_server_broker.**CacheControlledSiteTest**(*methodName='runTest'*)
Bases: twisted.trial._asynctest.TestCase

setUp()
Hook method for setting up the test fixture before exercising it.

testNoUICaching()

testUICaching()

Module contents

parlay.testing package

Subpackages

parlay.testing.unittest_mixins package

Submodules

parlay.testing.unittest_mixins.adapter module

Adapter Mixin. Use this mixin when doing unit tests to get access to a special unit test adapter that will allow you check on publish and subscriptions

class parlay.testing.unittest_mixins.adapter.**AdapterMixin**
Bases: object

Inherit from this Mixin to add a self.adapter that will let you hook in to publish and subscribe calls through self.adapter.published and self.adapter.subscribed

class AdapterImpl
Bases: *parlay.server.adapter.PyAdapter*

The actual adapter implementation for this mixin

publish(*msg*, *callback=None*)
:type msg dict

subscribe(*fn*, ***kwargs*)
:kwargs The topics and their values to subscribe to

adapter = <parlay.testing.unittest_mixins.adapter.AdapterImpl object>

parlay.testing.unittest_mixins.reactor module

class parlay.testing.unittest_mixins.reactor.**ReactorImpl**
Bases: *parlay.server.reactor.ReactorWrapper*

class parlay.testing.unittest_mixins.reactor.**ReactorMixin**
Bases: object

Inherit this class to get a self.reactor class that willa ct like a real reactor but give hooks for unit testing

reactor = <parlay.testing.unittest_mixins.reactor.ReactorImpl object>

Module contents

Submodules

parlay.testing.integrationtest module

Module contents

parlay.utils package

Submodules

parlay.utils.parlay_script module

Define a base class for creating a client script

class parlay.utils.parlay_script.**ParlayScript**(*item_id=None*, *name=None*, *_reactor=None*, *adapter=None*)

Bases: *parlay.items.threaded_item.ThreadedItem*

call_later(***kwargs*)

Calls function <fn> after <secs> seconds have passed.

Parameters

- **secs** – seconds to wait before calling function <fn>
- **fn** – function to call
- **args** – positional arguments that should be passed to <fn>
- **kwargs** – keyword arguments that should be passed to <fn>

Returns

cleanup(**args*)

Cleanup after running the script :param args: :return:

kill()

Kill the current script

on_message(*msg*)

Every time we get a message for us, this method will be called with it. Be sure to override this.

run_script()

This should be overridden by the script class

shutdown_broker()

parlay.utils.parlay_script.**start_script**(*script_class*, *engine_ip='localhost'*, *engine_port=8085*, *stop_reactor_on_close=None*, *skip_checks=False*, *reactor=None*)

Construct a new script from the script class and start it

:param script_class : The ParlayScript class to run (Must be subclass of ParlayScript) :param engine_ip : The ip of the broker that the script will be running on :param engine_port : the port of the broker that the script will be running on :param stop_reactor_on_close: Boolean regarding whether ot not to stop the reactor when the script closes (Defaults to False if the reactor is running, True if the reactor is not currently running) :param

skip_checks : if True will not do sanity checks on script (CAREFUL: BETTER KNOW WHAT YOU ARE DOING!)

parlay.utils.reporting module

parlay.utils.reporting.**log_stack_on_error**(*deferred*, *msg=''*)
Add an errback to the given deferred object to display the current stack trace as an aid for user debugging.

Parameters

- **deferred** – Twisted Deferred object to add errback to
- **msg** (*str*) – optional error message to display before the stack trace

Returns Deferred

Example Usage:

```
def myFunc():
    d = async_function_that_returns_deferred()

    # will show stack trace of myFunc and its callers
    d = log_stack_on_error(d, msg='')

    return d
```

parlay.utils.scripting_setup module

class parlay.utils.scripting_setup.**ThreadedParlayScript**(*item_id=None*, *name=None*, *_reactor=None*, *adapter=None*)
Bases: *parlay.utils.parlay_script.ParlayScript*

ready = False

parlay.utils.scripting_setup.**run_in_threaded_reactor**(*fn*)
Decorator to run the decorated function in the threaded reactor.

parlay.utils.scripting_setup.**setup**(*ip='localhost'*, *port=8085*, *timeout=3*)
Connect this script to the broker's websocket server.

Parameters

- **ip** – ip address of the broker websocket server
- **port** – port of the broker websocket server
- **timeout** – try for this long to connect to broker before giving up

Returns none

parlay.utils.scripting_setup.**start_reactor**(*ip*, *port*)

parlay.utils.twisted_log_observer module

class parlay.utils.twisted_log_observer.**LevelFileLogObserver**(*level=20*)
Bases: twisted.python.log.FileLogObserver

emit(*event_dict*)

log_dictionary(*event_dict*, *level*)
Format the given log event as text and write it to the output file.

@param event_dict: a log event @type event_dict: L{dict} mapping L{str} (native string) to L{object}

@param level: The event level to log.

Module contents

Submodules

parlay.constants module

parlay.enum module

parlay.enum.**enum**(**sequential*, ***named*)

parlay.errors module

exception parlay.errors.**TimeoutError**
Bases: exceptions.Exception

parlay.scripts module

parlay.scripts.**call_later**(*seconds*, *func*, **args*, ***kwargs*)

parlay.scripts.**close_protocol**(*protocol_id*)

parlay.scripts.**discover**(*force=True*)

parlay.scripts.**get_item_by_id**(*item_id*)

parlay.scripts.**get_item_by_name**(*item_name*)

parlay.scripts.**open**(*protocol_name*, ***kwargs*)

parlay.scripts.**open_protocol**(*protocol_name*, ***kwargs*)

parlay.scripts.**shutdown_broker**()

parlay.scripts.**sleep**(*time*)

parlay.scripts.**subscribe**(*fn*, ***kwargs*)

Module contents

class parlay.**WidgetsImpl**
Bases: object

Stub that will warn users that accidentally import or try to use widgets that it can only be used in the UI

parlay.**install_config**(*path*)
Install a config file to the parlay broker :param path: path to the config file :return:

parlay.**open_protocol**(*protocol_name*, ***kwargs*)

Sets up a protocol to be opened after the Broker initializes.

This function has the same effect as opening a new protocol from the browser-based user interface.

Parameters

- **protocol_name** (*str*) – name of protocol class to call the open method
- **kwargs** – keyword arguments to pass to the protocol's _open_ method

Returns none

Example Usage:

```
from parlay import open_protocol, start
open_protocol("ASCIILineProtocol", port="/dev/ttyUSB0", baudrate=57600)
start()
```

2.10 Specifications

These are the specifications that describe important interfaces of the Parlay system.

2.10.1 Parlay Standard Item Communications Specification

Parlay Standard Items are the most commonly used item type in the Parlay system. They support the *Command Streamable* and *Property* interfaces out of the box, and come with an intuitive automatic UI card in the UI. It is recommended that most items be sub-classed from the Parlay Standard Item.

JSON Message Format

All Parlay-defined keys are in CAPS. All custom keys are not.

All Parlay messages are JSON objects with two top-level keys:

- "TOPICS"
- "CONTENTS"

The value of each of these is a JSON object.

TOPICS Object

The Parlay-defined keys of the "TOPICS" object are as follows:

Key	Required?	Values
"TX_TYPE"	Yes	"DIRECT" or "BROADCAST"
"MSG_TYPE"	Yes	"COMMAND", "EVENT", "RESPONSE", "PROPERTY", or "STREAM"
"RESPONSE_REQ"	No (Default false)	Whether the sender requires a response to this message. true or false
"MSG_ID"	Conditional	Required only if a response is required ("RESPONSE_REQ": true). Unique Integer between 0 and 65535 (16-bit) from the sender. A Response to this message will use the same number.
"MSG_STATUS"	Conditional	See *Valid Values for MSG_STATUS key* section below. Required for message with MSG_TYPE = "RESPONSE".
"FROM"	Yes	The ID of the object this is message is from
"TO"	Conditional	For direct messages ("TX_TYPE": "DIRECT") only. The ID of the object this message is to.

Users can include their own keys in the TOPICS object. Those user-defined keys will have no effect on the Parlay Standard Item User Interface card.

Valid Values for MSG_STATUS key

The "MSG_STATUS" key in the "TOPICS" object is used for response or asynchronous messages. The following values are allowed for the "MSG_STATUS" key:

Value	Description
"ERROR"	Script will assert, UI will pop up an error.
"WARNING"	UI will pop up a warning. Message on console.
"INFO"	No action taken
"OK"	UI indicates successful completion
"PROGRESS"	Message successfully received, not completed. 0 or more PROGRESS messages can be sent

CONTENTS Dictionary

The other top level key in a Parlay JSON message is "CONTENTS". Most fields in the "CONTENTS" object are defined the discovery information provided by the Protocol. However, some fields are required based on the value of the MSG_TYPE field in the TOPICS object:

TOPICS/MSG_TYPE	CONTENTS Key	Required?	Value
"COMMAND"	"COMMAND"	Yes	Command identifier (string or number)
"COMMAND"	"COMMAND_NAME"	No	String name of command for display (default display is the Command identifier)
"RESPONSE"	"RESULT"	Conditional	Required if there is only one response parameter for the command. Should contain the response data.
"RESPONSE"	"<OUTPUT_PARAMETER>"	Conditional	For responses that have more than one piece of data a map of output parameters to data should be placed in CONTENTS.
"EVENT"	"EVENT"	Yes	Event identifier (string or number)
"EVENT"	"EVENT_NAME"	No	String name of event identifier for display (default display is the event identifier

If a message is "MSG_TYPE": "EVENT", or "MSG_TYPE": "RESPONSE" and "MSG_STATUS": "ERROR", then the Parlay Standard Item UI can display other information contained in the following fields:

Key	Required?	Value
"DESCRIPTION"	No	String for display of event or error response
"INFO"	No	List of additional informational strings

Messages with "MSG_TYPE": "STREAM" have the following fields. See the Stream interface for more detail.

Key	Required?	Value
"STREAM"	Yes	Stream ID
"RATE"	No	Sample Rate in Hz (0 to cancel sampling)
"VALUE"	No	Current value
"STOP"	No	True to stop streaming. (default False)

Messages with "MSG_TYPE": "PROPERTY" have the following fields. See the Property interface for more detail.

Key	Required?	Value
"PROPERTY"	Yes	Property ID
"ACTION"	Yes	"SET", "GET", or "RESPONSE". Errors will be handled with MSG_STATUS topic
"VALUE"	Conditional	Only required for "ACTION": "SET" or "ACTION": "RESPONSE" messages. The value to set the property to or the value that was retrieved

Item Discovery

Protocols must respond to Discovery requests with Discovery response messages. The format of a Discovery response message is defined elsewhere, but it includes a list of Item objects that have the following format.

Item ID Format

Item IDs are unicode strings that must be unique within the Parlay System. Uniqueness is not guaranteed by the Broker, but should be considered a fatal error by any system during discovery.

To ensure Item ID uniqueness, a hierarchical period-separated schema should be used. The first level should be the specific adapter type (e.g. 'python','Qt', etc). The specific sub-levels are left to the decision of the implementation, but should be detailed enough to ensure uniqueness and ease of management.

Some examples of ID:

- For an an item in Python: "python.promenade.LIMS" or "python.project_name.Linker"
- For an item on an embedded board: "ArmBoard.5.3ad2"

Item Object Format

Key	Required?	Value
"ID"	Yes	The system wide unique ID of the item. (*See Item ID Format*)
"NAME"	Yes	name of item
"TYPE"	No	< type of device, e.g.: "Waveform Generator", "Stepper Motor"... >
"TEMPLATE"	Yes	< e.g. 'sscom/STD_ITEM' >
"INTERFACES"	No	< list of interfaces that this item supports >
"CHILDREN"	No	< list of children Item objects >
"DATASTREAMS"	No	< list of DataStream objects (*see format below*) >
"PROPERTIES"	No	< list of Property objects (*see format below* >
"CONTENT_FIELDS"	Yes	< list of Field objects (*see format below*) that describe fields that will be in the CONTENTS field of messages from this item >
"TOPIC_FIELDS"	No	< list of Field objects (*see format below*) that describe fields that will be in the TOPICS field of messages from this item >

Property Object Format

Key	Required?	Value
"PROPERTY"	Yes	The property ID
"PROPERTY_NAME"	NO	The property name (Defaults to ID)
"INPUT"	Yes	"NUMBER", "STRING", "NUMBERS", "STRINGS", "ARRAY", "BOOLEAN", ["NUMBER(S)" || "STRING(S)" || "BOOLEAN"] Specifiying an array of types is used for specificying struct type order
"READ_ONLY"	No	Boolean, whether the property is read only, defaults to false
"WRITE_ONLY"	No	Boolean, whether the property is write only, defaults to false
"STREAMABLE"	No	Boolean, whether the property can be streamed, defaults to true

DataStream Object Format

Key	Required	Value
"STREAM"	Yes	The data stream ID
"STREAM_NAME"	No	The data stream name (Defaults to ID)
"UNITS"	No	Human readable string representing units of this data stream

Field Object format

Key	Required?	Value
"LABEL"	No	(label to show same as MSG_KEY if not defined)
"MSG_KEY"	Yes	< key passed with created message for this field >
"INPUT"	Yes	"NUMBER", "STRING", "NUMBERS", "STRINGS", "ARRAY", "DROPDOWN", "BOOLEAN" ["NUMBER(S)" || "STRING(S)" || "BOOLEAN"]
"REQUIRED"	No	If true, require the user fill out before sending command
"DEFAULT"	No	Default value for the input. If dropdown, then this will be the selected default
"HIDDEN"	No	If set to true, will hide the input from the user (i.e.: The default will be used as the value since the user can't change anything)
"DROPDOWN_OPTIONS"	Conditional	If input is a dropdown, must be a list of tuples
"DROPDOWN_SUB_FIELDS"	No	< list of Field objects>

Python Module Index

p

Index

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W