The Robot Plumbing Guide

The Hibike Team

2017-10-07

Contents

Chapter 1

Introduction

The driving principle behind PiE is running a robotics competition. This means many things: logistics, provinding mentorship and guidance, organizing events, providing an interface for programming the robot, and making the robot work. Keep in mind that this guide addresses only one aspect of the competition; plumbing is not everything. With that said, it is still a good idea to have a plumber on hand when pipes start to leak.

1.1 A Brief Overview of the Control Stack

For our purposes, there are five elements involved in controlling the robot.

1. Dawn: the main interface that students use. Takes inputs from the controller, and shows data and errors to students.

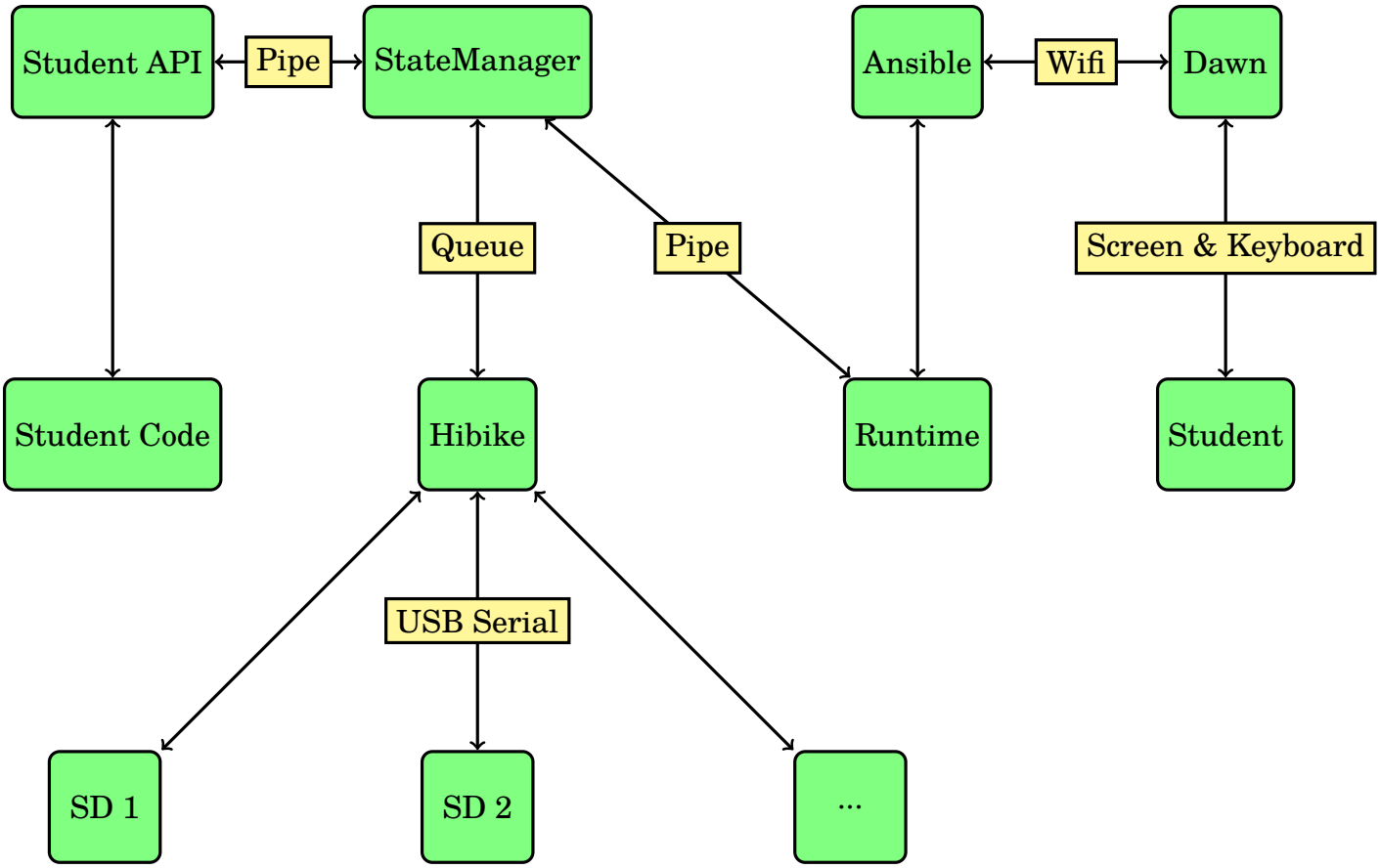
2. Ansible: the transmitter and reciever for data to and from Dawn.

3. Runtime: the program in charge of monitoring and communication. Sends and receives data from Dawn (via Ansible), as well as monitoring Hibike and StateManager.

4. StateManager: a data store. Provides a central place to store and retrive data from Hibike and Runtime.

5. Hibike: the program that communicates with sensors. Handles low-level protocol details.

1.2 A Diagram of the Stack


Student API
Pipe
StateManager
Ansible
Wifi
Dawn
Queue
Pipe
Screen & Keyboard
Student Code
Hibike
Runtime
Student
USB Serial
SD 1
SD 2
...

Chapter 2

Hibike

Hibike is in charge of low-level sensor communication, and all the gory details that are involved. It is divided into two modules:

- `hibike_message.py`

  The low-level details of the Hibike communications protocol. Handles things like encoding into packets and checksums.

- `hibike_process.py`

  The "supervisor" of sensors. Communicates with sensors, sends data to runtime, and reacts to orders from StateManager.

Alongside these main modules, there are various testing modules:

- `virtual_device.py`

  Virtual devices, for testing Hibike's read and write capabilities.

- `hibike_tester.py`

  A test module that wraps a Hibike process with a nicer interface.

2.1 Design Choices and Explanations

Hibike is packet-based and works over USB, specifically serial.

Why USB? In previous years, a custom communications protocol was used, over custom cables. This provided the advantage of real-time communications and configurability; however, debugging was very difficult, and developing for smart sensors more so. The Beaglebone doesn't provide real-time guarantees anyways, and USB provides cheap, commodity cables and well-tested software.

Why packets? Most protocols are packet-based, and serial is not a packet-based communications link. Packets make it easy to tell when data begins and ends.

Why serial? Arduino provides a well-tested serial library, so communication with any Arduino is easy over serial. In addition, implementing and using serial is relatively simple compared to many other types of USB connection, such as HID (used by keyboards and mice). Some flexibility and speed is lost as a result of this abstraction, but much more is gained in ease of use.

2.2 The Hibike Protocol

2.2.1 A Quick Introduction

We make a few starting assumptions concerning the endpoints of communication: the device controlling a sensor is a Smart Device (SD), and a Beaglebone Black (BBB) is used as the central control board of each robot and thus is in charge of communicating with each Smart Device. These two communicate with each other via serial: the Beaglebone running pySerial and the Smart Device by using the built-in serial library. As each Smart Device communicates with the Beaglebone on its own separate port, we conveniently have no need to worry about any race conditions or other problems arising from concurrency.

Hibike abstracts every Smart Device as a set of (parameter, value) pairs. The hibike protocol supports three ways of interacting with these parameters:

- Subscribing to regular updates of specific paramters
- Polling specific parameters
- Writing to specific parameters

Refer to Section 7 for an outline of the general behavior of the Hibike protocol.

2.2.2 Message Format

All messages have the relatively simple structure of Message ID, Payload, and Checksum as depicted below. A more complete description of each field is given below the diagram.

Message ID (8 Bits)	Payload Length (8 Bits)	Payload (Length Varies)	Checksum (8 Bits)

- Message ID
  - An 8-bit ID specifying the type of message.
  - More information about this field in the following sections.
- Payload Length
  - An 8-bit unsigned integer, specifying the number of bytes in the payload.

- Payload
  - Varies depending on the type of message sent.
- Checksum
  - An 8-bit checksum placed at the end of every message.]
  - The current scheme XORs every other byte in the message.

2.2.3 UID Format

Each smart device is assigned an 88-bit UID with the following fields.

Device Type (16 Bits)	Year (8 Bits)	ID (64 bits)

- Device Type
  - 16-bit ID specifying the Type of a Smart Device
  - Device types are enumerated in Section 5
- Year
  - 8-bit ID corresponding to the competition year that the Smart Device was manufactured for.
  - The 2015-2016 season will correspond to 0x00
- ID
  - Randomly generated 64-bit number that will uniquely identify each Smart Device within a specific device type and year.
  - With 64-bit IDs, the probability of a hash collision with 1000 of 1 type of device per year is roughly 0.05%

2.2.4 Parameters and Bitmaps

- Hibike abstracts every smart device as a set of parameters that map to values
- Each smart device contains some number of paramaters, which can be read/written to.
- Parameters can have many types. The following types are supported:
  - `bool`
  - `uint8_t`
  - `int8_t`

  - `uint16_t`
  - `int16_t`
  - `uint32_t`
  - `int32_t`
  - `uint64_t`
  - `int64_t`
  - `float`
  - `double`

  CAUTION: Arduino's doubles are only 4 bytes long (same as a float), so an Arduino's Double is the same as python's Float. Do not use this type unless your arduino is actually cranking out 8 bytes.

- Some paramaters are read only, some are write only, and some support both.
- A config file will describe the paramaters for each Device Type (name, type, permissions).
- Some packets encode sets of parameters in the form of bitmaps.

Params (16 bits)	Value 0 (Optional and Variable)	... ...	Value 15 (Optional and Variable)

  - Params - 16-bit bitmap describing a set of parameters. The nth bit of Params, where the LSB is the 0th bit, references the nth paramater of a device.
  - Value[0-15] - DeviceWrite and DeviceData send actual values for each param in Params. The value field for param n will only be present if bit n in Params is set. The size and type of each value field depends on param number and device type, and is described in a config file.

2.2.5 Message Types

ID	Message Type
0x10	Ping
0x11	Subscription Request
0x12	Subscription Response
0x13	Device Read
0x14	Device Write
0x15	Device Data
0x16	Device Disable
0x17	Heartbeat Request
0x18	Heartbeat Response
0xFF	Error

2.2.6 Device Types

ID	Sensor	Param Number	Param Name	Param Type	Read?	Write?
0x00	LimitSwitch	0	switch0	bool	yes	no
		1	switch1	bool	yes	no
		2	switch2	bool	yes	no
0x01	LineFollower	0	left	float	yes	no
		1	center	float	yes	no
		2	right	float	yes	no
0x02	Potentiometer	0	pot0	float	yes	no
		1	pot1	float	yes	no
		2	pot2	float	yes	no
0x03	Encoder	0	rotation	int16_t	yes	no
0x04	BatteryBuzzer	0	is_unsafe	bool	yes	no
		1	calibrated	bool	yes	no
		2	v_cell1	float	yes	no
		3	v_cell2	float	yes	no
		4	v_cell3	float	yes	no
		5	v_batt	float	yes	no
		6	dv_cell2	float	yes	no
		7	dv_cell3	float	yes	no
0x05	TeamFlag	0	mode	bool	yes	yes
		1	blue	bool	yes	yes
		2	yellow	bool	yes	yes
		3	led1	bool	yes	yes
		4	led2	bool	yes	yes
		5	led3	bool	yes	yes
		6	led4	bool	yes	yes
0x06	Grizzly					
0x07	ServoControl	0	servo0	float	yes	yes
		1	servo1	float	yes	yes
0x08	LinearActuator					
0x09	ColorSensor					
0x0A	YogiBear	0	duty_cycle	float	yes	yes
		1	pid_pos_setpoint	float	no	yes
		2	pid_pos_kp	float	no	yes
		3	pid_pos_ki	float	no	yes
		4	pid_pos_kd	float	no	yes
		5	pid_vel_setpoint	float	no	yes
		6	pid_vel_kp	float	no	yes
		7	pid_vel_ki	float	no	yes
		8	pid_vel_kd	float	no	yes
		9	current_thresh	float	no	yes
		10	enc_pos	float	yes	yes

ID	Sensor	Param Number	Param Name	Param Type	Read?	Write?
		11	enc_vel	float	yes	no
		12	motor_current	float	yes	no
		13	deadband	float	yes	yes
0x0B	RFID	0	id	uint8_t	yes	no
		1	detect_tag	uint8_t	yes	no
0x10	DistanceSensor					
0x11	MetalDetector					
0xFFFF	ExampleDevice	0	kumiko	bool	yes	yes
		1	hazuki	uint8_t	yes	yes
		2	sapphire	int8_t	yes	yes
		3	reina	uint16_t	yes	yes
		4	asuka	int16_t	yes	yes
		5	haruka	uint32_t	yes	yes
		6	kaori	int32_t	yes	yes
		7	natsuki	uint64_t	yes	yes
		8	yuko	int64_t	yes	yes
		9	mizore	float	yes	yes
		10	nozomi	double	yes	yes
		11	shuichi	uint8_t	yes	no
		12	takuya	uint16_t	no	yes
		13	riko	uint32_t	yes	no
		14	aoi	uint64_t	no	yes
		15	noboru	float	yes	no

2.2.7 Error Types

Status	Meaning
0xFD	Unexpected Packet Delimiter
0xFE	Checksum Error
0xFF	Generic Error

2.2.8 Message Descriptions

1. Ping: BBB pings SD for enumeration purposes. The SD will respond with a SubscriptionResponse packet.

   Payload format:

Empty (0 bits)

   Direction: BBB ⇒ SD

2. SubscriptionRequest: BBB requests data to be returned at a given interval.

- Params is a bitmap of paramaters being subscribed to.
- The SD will respond with a Sub Response packet with a delay and bitmap of params it will acutally send values for, which may not be what was requested, due to nonexistent and write-only parameters.
- If too many parameters are subscribed to, the Smart Device may have to send multiple DeviceData packets at each interval.
- A delay of 0 indicates that the BBB does not want to receive data.
- Non-zero delay with 0 Params will still subscribe to empty updates!!!

Payload format:

Params (16 bits)	Delay (16 bits)

Direction: BBB $\Rightarrow$ SD

3. SubscriptionResponse: SD sends (essentially) an ACK packet with its UID, params subscribed to, and delay.

   Payload format:

Params (16 bits)	Delay (16 bits)	UID (88 bits)

   Direction: BBB $\Leftarrow$ SD

4. Device Read: BBB requests some values from the SD.

   - The SD should respond with DeviceData packets with values for all the readable params that were requested.
   - If all the values cannot fit in one packet, multiple will be sent.

   Payload format:

Params (16 bits)

   Direction: BBB $\Rightarrow$ SD

5. Device Write: BBB writes attempts to write values to some parameters.

   - The SD should respond with a DeviceData packet describing the readable params that were actually written to.
   - The protocol currently does not support any ACK for write only params.

   Payload format:

Params (16 bits)	Value 0 (Optional and Variable)	... ...	Value 15 (Optional and Variable)

Direction: BBB $\Rightarrow$ SD

6. Device Data: SD sends the values of some of its paramters.

   - This can occur in response to a DeviceWrite/DeviceRead, or when the interval for a SubscriptionResponse occurs.

   Payload format:

Params (16 bits)	Value 0 (Optional and Variable)	... ...	Value 15 (Optional and Variable)

   Direction: BBB $\Leftarrow$ SD

7. Heart Beat Request: BBB/SD requests SD/BBB heartbeat response for connectivity purposes.

   - This message pathway is a two way street, both BBB and SD can send requests and send responses to the other
   - Upon receiving a Heart Beat Request, a Heart Beat Response message should be immediately sent back
   - Payload is currently unused, but can be used for future functionality in keeping track of individual heartbeat requests and responses (for latency purposes)

   Payload format:

ID (8 bits)

   Direction: BBB $\Leftrightarrow$ SD

8. Heart Beat Response: Sent in response to a Heart Beat Request

   - This message pathway is a two way street, both BBB and SD can receive requests and send responses to the other
   - Should only be sent upon receiving a Heart Beat Request
   - Payload is currently unused, but can be used for future functionality in keeping track of individual heartbeat requests and responses (for latency purposes)

   Payload format:

ID (8 bits)

Direction: BBB ⇔ SD

9. Error Packet: Sent to indicate an error occured.

   - Currently only used for the BBB to log statistics.

   Payload format:

Error Code (8 bits)

   Direction: BBB ⇐ SD

2.3 The Hibike Process

`hibike_process.py` handles communication with sensors and StateManager. Initialization and running happen in this series of steps:

1. Hibike asks the operating system for a list of serial ports that match those used by Arduinos.
2. It verifies that these ports are working, and opens them for communication.
3. Hibike checks that each port contains a smart sensor by pinging it.
4. Each device is given a reading thread, a writing thread, and a write queue.
5. Alongside the device threads, a read batching thread and a cleanup thread are started. The batch thread sends all reads to StateManager at certain intervals. The cleanup thread removes disconnected devices.
6. Hibike tells all devices to stop sending data. Once the devices respond, StateManager subscribes to each device for each of its parameters.
7. Afterwards, Hibike receives and reacts to commands from StateManager.

Hibike devices are represented as a named tuple, with read and write threads, a serial port, a write queue, and an instance ID, which is generated randomly when the device starts up.

2.3.1 An Explanation of the Architecture

Each part of the structure of the Hibike process has its place, and has been designed in deliberately. If threads are used, it is usually because of blocking I/O.

Reader and Writer Threads Reading from a serial connection using nonblocking communication means that it is easily possible to receive an incomplete packet even if the communications link is reliable; thus, we use blocking reads. We don't want to stop reading from every other sensor if one is taking a while, so we use threads to periodically switch between sensors.

The Write Queue If a device sends a heartbeat request, we want to respond quickly without going all the way to StateManager. Thus, we have a write queue, allowing both the main thread and the read thread to write packets.

The Batch Thread Inter-thread communication is quick, but inter-process communication is slow. Consequentially, Hibike threads can quickly talk to each other, but talking to StateManager, which runs in a separate process, is quite time-consuming. Instead of sending any device data as soon as it is read, it is easier to bundle many reads up and send them all at once. The batch thread is responsible for this.

The Cleanup Thread When a device disconnects, its resources remain in use. The read and write threads sit idle, and the serial port remains open. This can cause problems if a new device appears on an existing serial port, so it is useful to clean up the resources. At the same time, it can take up to 30 seconds to close a serial port, which is unacceptably long for Hibike to wait. Therefore, a separate thread is put in charge of such things.

The Hotplug Thread In order to detect new devices, something has to scan serial ports and check which contain smart sensors. This could be done in the main loop, but detection can take hundreds or even thousands of milliseconds.

Instance IDs Devices already have an associated UID, that is (hopefully) unique to every device. Why is it necessary to have yet another unique identifier? The answer lies with hotplugging: it is possible for a device to disconnect and then reconnect. In this case, it still has the same UID, so we have no way of telling whether it has reconnected. The solution is to associate an ID with a particular connection to the device, so that the device will have a different instance ID if it reconnects.

2.4 Smart Devices

There are a few types of smart device, each with its own responsibility. Smart devices are based on the Arduino platform; we are using Arduino Pro Micros because of their relative cheapness ($4/unit) and power. The firmware for these devices is written in C++.

- LimitSwitch

The name of this sensor is pretty self-explanatory. It's a limit switch (or two, or three), that is on when pressed.

- LineFollower

  This is a line-following sensor, that detects the brightness of the floor beneath it. There are three sensors: left, middle, and right.

- BatteryBuzzer

  This device monitors the battery voltage and cell balances, making sure that the battery doesn't get dangerously imbalanced or low. It displays the cell voltages on an LED display, and beeps when there are problems. Its functionality may be supplanted in the future by a dedicated chip.

- ServoControl

  The servo controller controls the position of a servo, in the range [-1, 1]. The position can be read or written to.

- YogiBear

  This is the new motor controller, put into place after the repeated demise of many a Grizzly. It uses a COTS motor controller, and has the ability to perform PID (supposedly). If Hibike does not respond to heartbeats, it disables itself automatically.

- RFID

  This sensor is unfortunately slow; sometimes, it will not read, and the read must be delayed for a cycle. It is also very close range; usually, about 0.5in is the maximum.

2.5 Hotplug

Smart sensor hotplug has been a highly requested feature for some time now. There are two halves to hotplug: adding new devices, and disconnecting old ones. Both sides of the implementation are based on polling – periodically checking whether new devices have been connected, or old ones removed.

2.5.1 Adding New Devices

Every period of `HOTPLUG_POLL_INTERVAL` seconds, the hotplug thread wakes up and goes looking for new devices. It does this using the existing `get_working_serial_ports`, excluding any ports that are already attached to existing devices. Afterwards, it scans these serial ports for smart sensors, and adds them to the list of devices, spinning up threads and queues as appropriate.

2.5.2 Removing Disconnected Devices

After adding any new devices, we go on the lookout for old devices that have been disconnected. When a device is disconnected, its read and write threads crash, and we receive a message in `error_queue`. If we have only received one message from a device, we assume that the device may only have been temporarily disconnected, and resend the message for the next iteration. If the existing device has a different instance ID, then we assume that the device has reconnected. Otherwise, we tell the cleanup thread to clean the device.

2.6 Testing

Hibike provides a few options for testing protocol compatibility and generating data. In general, there is no substitute for testing on real hardware, but there are at least a few tests.

2.6.1 Latency

Full-stack latency testing exists in the form of the `TIMESTAMP_UP/DOWN` commands, which are passed from Dawn to Ansible to StateManager, to Hibike, and back up. At the end, Dawn prints the timestamps that each level sent.

2.6.2 `hibike_tester.py`

This module acts as a fake StateManager, spawning its own copy of `hibike_process`. You can use this to verify that Hibike is sending device data to state manager, and that it reacts appropriately to commands sent to it.

2.6.3 `virtual_device.py`

This implements virtual devices, which communicate with Hibike through a virtual serial port. It is meant to be used with `spawn_virtual_device.py` to create virtual devices and send data on them. It can be used to test Hibike's ability to respond to devices.

2.6.4 Unit Tests

Unit tests for some of Hibike's modules are located in the `unit_tests` directory, and make use of the `unittest` Python module. Each function that is tested has its own class, and a few test cases. These tests should not be considered comprehensive.

2.7 Code Samples

2.7.1 hibike_tester.py

Basic Usage

```
"""
Example code to print the UID of every sensor attached to the
robot.

Author: Brose Johnstone
"""
from hibike_tester import Hibike
import hibike_message as hm

robot = Hibike()
robot.enumerate()

while True:
    packet = robot.state_queue.get()
    if packet.get_message_id() == hm.MESSAGE_TYPES["SubscriptionResponse"]:
        # Get the UID
        _, _, uid = hibike_message.parse_subscription_response(packet)
        print(uid)
```

Sending and Receiving Packets

```
"""
Request sensor data from a single device.

Author: Brose Johnstone
"""
from hibike_tester import Hibike
import hibike_message

# The UID of the device to read from
DEVICE_UID = # XXX: fill in yourself
DEVICE_ID = hibike_message.uid_to_device_id(DEVICE_UID)
# The list of parameters to read
DEVICE_PARAMS = # XXX: Fill this in yourself

robot = Hibike()
robot.read(DEVICE_UID, DEVICE_PARAMS)

while True:
```

```
    packet = robot.state_queue.get()
    if packet.get_message_id() == hibike_message.MESSAGE_TYPES["DeviceData"]:
        decoded = hibike_message.parse_device_data(packet, DEVICE_ID)
        print(decoded)
        break
```

Subscribing to a Device

```
"""
Subscribing to a device to receive regular updates.

Author: Brose Johnstone
"""
from hibike_tester import Hibike
import hibike_message

DEVICE_UID = ...
DEVICE_ID = hibike_message.uid_to_device_id(DEVICE_UID)
DEVICE_PARAMS = ...
SUBSCRIPTION_DELAY = ...

robot = Hibike()
robot.subscribe(DEVICE_UID, SUBSCRIPTION_DELAY, DEVICE_PARAMS)

while True:
    packet = robot.state_queue.get()
    if packet.get_message_id() == hibike_message.MESSAGE_TYPES["DeviceData"]:
        decoded = hibike_message.parse_device_data(packet, DEVICE_ID)
        print(decoded)
```

2.7.2 Identifying Serial Ports and Smart Sensors

```
"""
Identify which serial ports contain devices, and print the mapping
of serial ports to UIDs.

Author: Brose Johnstone
"""
from hibike_process import get_working_serial_ports, identify_smart_sensors

ports, port_names = get_working_serial_ports()
devices = identify_smart_sensors(ports)

for (port_name, uid) in devices.items():
    print(port_name, uid)
```

2.7.3 Working with UIDs

```
from hibike_message import *

# Fill this in with whatever UID you are working with
MY_UID = ...

# Print the type of the device, based on the UID.
print(uid_to_device_name(MY_UID))
# Turn the UID into a device ID.
DEVICE_ID = uid_to_device_id(MY_UID)
# Get possible parameters for the device
print(all_params_for_device_id(DEVICE_ID))
# Check if a parameter is readable (or writeable)
print("Is parameter {} readable? {}".format("XXX", readable(DEVICE_ID, "XXX"))
print("Is parameter {} writeable? {}".format("XXX", writeable(DEVICE_ID, "XXX"))
```

Chapter 3

Runtime, StateManager, and Ansible

3.1 Runtime's Architecture

Runtime supervises other processes and handles errors as they occur. It is essentially a state machine that reacts to messages from Dawn and student code. If any process crashes, Runtime would (ideally) be able to restart it seamlessly, although this doesn't happen yet.

Why Pipes over Queues? A pipe is more limited than a queue in every respect; it is single-consumer, single-producer, as opposed to multi-producer, multi-consumer. However, what it lacks in flexibility, it makes up for in speed. There is a noticeable difference when using pipes versus queues. In a scenario where the flexibility is unnecessary, pipes win.

3.1.1 Runtime Steps

1. Spawn StateManager, Hibike, and UdpReceiver.
2. Run the state machine.
   (a) Check if E-Stop was triggered.
   (b) Check if we have restarted too many times.
   (c) React to things in the bad things queue.

3.1.2 The `badThingsQueue`

This unfortunately-named queue allows every process to send messages directly to runtime. Typically, this is used as an error reporting mechanism when an exception is encountered, so that runtime will at least know about the cause of a failure. It is also used as a trigger to switch the robot from teleoperated to autonomous mode.

3.1.3 `runtimeUtil` and IPC

Communication between processes happens by passing messages through StateManager. Each process is given access to StateManager's message queue, and StateManager can pass messages back to each process through pipes. In order to make sure that processes can understand each other, there is a set of several constants located in `runtimeUtil.py`, defining the messages that a process is expected to send and receive.

3.2 StateManager

StateManager is, in essence, a large key-value store. It is used as a centralized place to store state for each process on the robot, and, indirectly, as a way to communicate between processes.

3.2.1 Architecture

StateManager consists of a large number of methods and a few message queues/pipes. It receives messages from Hibike, Runtime, Ansible, and Student Code, and invokes one of its functions based on who sent the message, and which message was received. Every time a key is added to storage, a timestamp is stored. This functionality will likely be removed in the future.

3.2.2 Robot State

The internal state of the robot is stored in a dictionary. Each key represents one piece of the robot's state.

- `studentCodeState` represents the current mode of the student code: teleoperated, autonomous, or emergency stop
- `dawn_addr` stores information about the IP address of Dawn.
- `gamepads` stores information about the state of any attached gamepads, for teleoperated mode.
- `hibike` stores information about the state of every smart device attached to the robot. If a smart device is disconnected, it disappears from this dictionary.

3.3 Ansible

3.3.1 Introduction and Overview

This page will detail how Runtime communicates with Dawn. The code is contained primarily in `Ansible.py`, with relative protos stored in ansible-protos.

There are two forms of communication with Dawn: UDP and TCP. UDP is used for constant data transfer where it is not critical to not lose packets. TCP is used to send data where it is critical to not lose information. Communication with Dawn happens in three processes, which are spawned by `runtime.py`. These three processes are `UDPSendClass`, `UDPRecvCLass`, `TCPClass`. `UDPSendClass` spawns two threads, one for packaging and one for sending. `UDPRecvClass` does not spawn threads, and receives and unpackages in succession. `TCPClass` spawns two threads, one for sending and one for receiving.

See Dawn Runtime Network Protocol for specific networking details, and Google Protobuf for information on how to use protobufs in python.

3.3.2 Starting the Processes

All processes are started by `runtime.py` via the `spawn_process` function. Initially, on startup, only the `UDPRecvClass` class is spawned. This is because runtime needs to learn the IP of Dawn, and can only do so after receiving one packet. After determining the IP of Dawn and it is stored into `StateManager`, `BAD_EVENTS.NEW_IP` is triggered which now spawns `UDPSendClass` and `TCPClass`. Only one Dawn instance can be connected at a time. Whenever Dawn disconnects, `BAD_EVENTS.DAWN_DISCONNECTED` is triggered which terminates all the processes and then respawns just `UDPRecvClass`. This is the same as the initial state, and it repeats the entire process.

3.3.3 UDPRecvClass

This class runs as a single process. It receives the robot state (`IDLE`, `AUTONOMOUS`, `TELEOP`, `ESTOP`), the team color, and gamepad data.. If there doesn't currently exist an IP stored (this is the first time receiving from Dawn), the IP will also be saved and the other two communication processes will be spun up. If the received state is different from the current state, a command will be sent to `StateManager.py` via the `BadThingsQueue` to switch student code state. If the team flag color has not been set, a command (`SM_COMMANDS.SET_TEAM`) will be sent to `StateManager.py` to set the team flag to the appropriate color. Then, it takes the unpackaged data containing the gamepad data, and sends it to `StateManager.py`.

3.3.4 UDPSendClass

This class runs as two threads in a single process. One process constantly receives data from `StateManager`, and packages up via protobufs all the sensor information, and the current robot state. It then stores the data into a
`TwoBuffer` object which is shared by both processes. The second process takes the packaged data from the `TwoBuffer`, and sends it to Dawn viw UDP.

3.3.5 TCPClass

This class runs as two threads in a single process, one to send and one to receive. Both use the same TCP connection. The sender sends console logs to Dawn, as well as acknowledges that runtime is ready for student code upload. The receiver receives the command that student code is going to be uploaded. Furthermore, when the TCP connection is broken (as caught by an `except`), this indicates that Dawn has disconnected and `Ansible` state is reset to idle.

3.4 Student Code/Student API

Student code itself runs in a separate process isolated from runtime, so that it is not possible to deadlock runtime from student code. In addition, signal handlers are installed to ensure that the code will stop working if stalled for too long. The API communicates with StateManager through a pipe.

3.5 Testing

Currently, Runtime's tests are located in `studentCode.py`. What exists now is pretty basic, consisting mainly of fetching from and storing into StateManager.

Chapter 4

Glossary

- Beaglebone Black (BBB) – a single-board computer running Ubuntu Linux. More information at `http://beagleboard.org/black`.
- Smart Sensor or Smart Device (SD) – a sensor, such as a switch or RFID reader, that is controlled by an attached Arduino.