

# PiBrain Documentation

Andrew M. Evans

July 2021

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
1.1	Description . . . . .	2
1.2	Purpose . . . . .	2
1.3	Contact information . . . . .	2
<b>2</b>	<b>Software</b>	<b>3</b>
2.1	Bash scripts . . . . .	3
2.1.1	Clearing output cache . . . . .	3
2.1.2	Automated startup . . . . .	3
2.2	State machine . . . . .	4
2.2.1	Imports . . . . .	4
2.2.2	State machine class . . . . .	4
2.2.3	write_to_log . . . . .	4
2.2.4	idleTaskCheck . . . . .	6
2.2.5	idle_state . . . . .	6
2.3	Main . . . . .	7
2.3.1	Manual mode . . . . .	7
2.3.2	Imports . . . . .	7
2.3.3	bcolors . . . . .	7
2.3.4	IO_Devices . . . . .	7
2.3.5	Tasks . . . . .	8
2.3.6	Pin definitions . . . . .	9
2.3.7	Function . . . . .	9
<b>3</b>	<b>Hardware</b>	<b>9</b>
3.1	Raspberry pi . . . . .	9
3.2	Deposition machine . . . . .	9
3.2.1	PLC . . . . .	9
<b>4</b>	<b>Installation</b>	<b>11</b>

<b>5</b>	<b>Future work</b>	<b>11</b>
5.1	Combination of bash scripts with python . . . . .	11
5.2	write_to_log rework . . . . .	11
5.3	Idle exit . . . . .	11
5.4	Main file split . . . . .	11
5.5	Deposition controller pi . . . . .	11
5.6	Web interface . . . . .	11
5.7	GUI interface . . . . .	11
5.8	Automatic mode . . . . .	11
5.9	IO_Device class change . . . . .	11

# 1 Introduction

This document aims to give a complete outline of both the software and hardware used in the PiBrain v0.1-alpha. The documentation will also act as an installation guide, and a place to detail potential future projects. The L<sup>A</sup>T<sub>E</sub>X source code for this PDF will be included in the PiBrain repository on GitHub, it is highly recommended that any future work on PiBrain be document here.

## 1.1 Description

PiBrain is a raspberry pi operated replacement for PLC on the MK-VII thermal evaporator system. The software is written in primarily python 3.8.0, with some auxiliary scripts written in bash. The hardware is a simple 24V to 3.3V logic conversion using ISOs and relays.

## 1.2 Purpose

The main purpose of PiBrain is to provide an easily modifiable platform for controlling the MK-VII thermal evaporator system. Section 5 focuses on future applications/ideas for what the system could be capable of. The primary focus of this document will be establishing how the hardware was built, and how the software was written with respect to the manual mode.

## 1.3 Contact information

To contact the original creator (Andrew) please use the following resources:

- Email: [evansa@sonoma.edu](mailto:evansa@sonoma.edu) (only good till 2023)
- Alternate email: [andrew.m.evans1989@gmail.com](mailto:andrew.m.evans1989@gmail.com)
- Discord: Andrew Evans#4366

Responses wont be instantaneous, but I will try and get around to it!

## 2 Software

This section will detail all current pieces of the code along with idea behind the organization and structure of the code. The code is broken up into three categories: bash scripts, state machine, and main. The general idea of the code is that the bash scripts do the operating system level work (automated startup, file clean up, etc see Section 2.1), the state machine does the “organizational” work of the software (running the main logic tree which includes the idle loop see Section 2.2), and lastly main which contains the hardware interfacing code (see Section 2.3). There are some aspects of this layout which could<sup>1</sup> be changed in the future (see Section 5).

### 2.1 Bash scripts

Currently there are two bash scripts: `clear_run_cache.sh`, and `startCode.sh`. Both scripts are incredibly simple and act as a way of automatically running system commands

#### 2.1.1 Clearing output cache

The first script is responsible for clearing the outputfile cache. During run time the code will produce an outfile file which logs changes while the code is running. For example if the vent command is called then it will log the vent command along with a time stamp, see Figure 1.

```
9 2021-07-21 19:35:55.035873 Mechanical pump didn't complete, conditions not met, returning to queue
10 2021-07-21 19:36:00.055460 venting system completed, returning to queue
11 2021-07-21 19:36:29.168215 Shutting system down
```

Figure 1: Example log file

After the runs the bash script will clear this file **NEED TO ADD OPTION FEATURE FOR THIS**. default setting will be to clear the run cache after run.

#### 2.1.2 Automated startup

The other script, `startCode.sh`, is responsible for starting the code on the boot up of the raspberry pi. **NEED TO ADD FILE TO BOOT DIRECTORY**. The hope is that SSH connection to the pi would be minimal, and that in most of the use cases the raspberry pi could automatically start and clear run caches without the need for human intervention, see Section 3.1.

---

<sup>1</sup>most likely should be changed

## 2.2 State machine

Statemachine.py is the organizational backbone of the code fig 2. Any defined command should run through the state machine if possible, however the definitions should occur outside the state machine<sup>2</sup>

### EDIT PICTURE OF STATE MACHINE

### 2.2.1 Imports

The imports are used for getting the date and time for the log file, and for pausing the script. The main reason for wanting a script pauses is that the main loop of the state machine is effectively an infinite loop. Without the pause feature the loop runs much faster than a user could ever hope to make inputs. Documentation for the modules: [datetime](#), [time](#)<sup>3</sup>

### 2.2.2 State machine class

The state machine itself is defined as an object in the code. It has two attributes: `outputfile`, and `inputfile`. These names make the purpose pretty obvious, but for those who haven't caught on, it's for setting the input and output file paths. The output file is what the log is written into, and the input file is designed for simulating the machine. The input feature will likely be changed or removed before the final draft of this document **YOU HEAR THAT? YOU SHOULD EDIT THIS OUT LATER**. The state machine itself has three methods: `write_to_log`, `idleTaskCheck`, and `idle_state`. These methods will be discussed in more detail below.

### 2.2.3 write\_to\_log

This method looks at the return codes thrown by the different tasks and using this determines what message to print into the log file. Tasks return a three-tuple of which the first two elements are booleans, and the last is a string. The general form of the tuple is:

(Expected exit, Task completed, Name of task)

Table 1 shows what the different combinations of trues and falses yield.

- **Key** to table 1:
  - OK -the system completed the task
  - NC -the task didn't change
  - NA -not applicable (combination not set)
  - FAIL -an error has occurred

---

<sup>2</sup>Again, this is something that was not done super well and will be addressed in Section 5

<sup>3</sup>[time](#) and [datetime](#) are hyperlinks, click to see the documentation

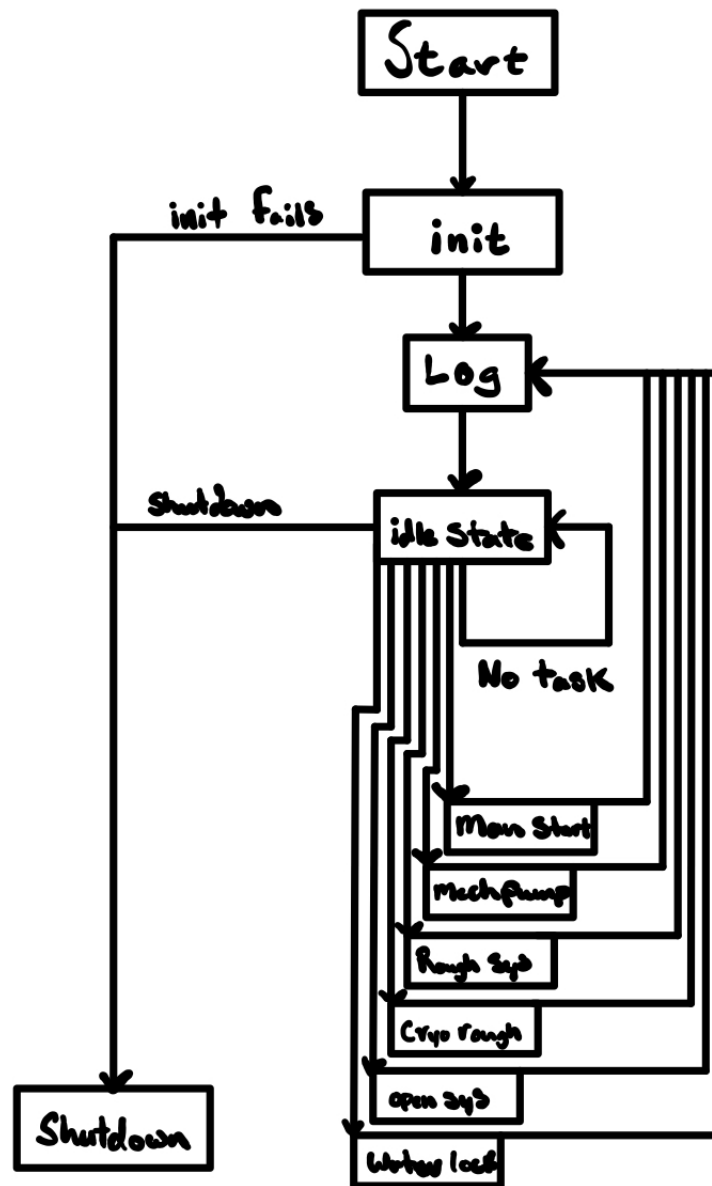


Figure 2: State machine logic

While this system works it is definitely another point in the code which should be reworked in the future, see Section 5.

	T	F
T	OK	NC
F	NA	FAIL

Table 1: Return statement logic

#### 2.2.4 idleTaskCheck

This method simply compares the last task with the current task looking to see if there had been a change in the task’s state. If the task’s state had changed then the function writes this to the log file. This was designed so that the log file would only report changes in task, reducing the overall space used on the raspberry pi.

#### 2.2.5 idle\_state

the idle\_state method is the heart of the state machine. The method is responsible for all code that is updated every loop. There are eight inputs to this method (excluding the self input). All the inputs that start with “task” are the actual logical functions defined in main passed into the method. The update\_pins input is the pin reading function defined in main called readPinsLoop. The first variable (called delay) inside the idle loop gives the ability to set the delay on the actions in the loop in seconds. During the loop the state machine pauses and waits one second between each prompt. This is to ensure that the code is not overworking the raspberry pi. After the delay is set the code initializes all the “task#\_last”’s this is done so that the idleTaskCheck method will work on the first loop, and so that the code will write all of task’s states to the log file upon startup.

After the initialization of the idle loop the variable machine\_on is set to true and the code enters the while loop. This loop is infinite and will only close when machine\_on gets set to false. This can happen in only two cases.

**Case 1:** If any of the tasks return statements throw a first element false, this is defined as an emergency halt. For more information on the tuple edit code system see Section 2.2.3.

**Case 2:** If the stop pin **MIGHT CHANGE** is called.

Case 1 can only be called if the update\_pins function doesn’t return a false. False is returned from the function if case 2 will happen. This was designed this way so that if a stop code is called it wont try and run all the tasks. This method could be improved, see Section 5.3.

## 2.3 Main

Main is where the code is run from and where the definitions of the pins and processes are. If additional logic is added to the code the definitions should be put in main. In the future the code written in main could be split into two separate files so that the definitions are done in another file see Section 5.4 for more detail.

### 2.3.1 Manual mode

The code currently supports a manual mode. This means that the user must toggle switches on the actual deposition machine itself, which sends signals to the raspberry pi, and then finally the pi decides if it is “safe” to turn on the called process. This means that without explicitly telling the pi what to do nothing will happen. This system is capable of being fully automatic, for more detail on this project please see Section 5.8.

### 2.3.2 Imports

This piece of code currently imports the following packages: time, datetime, StateMachine, and os. For more information on time, and datetime see Section 2.2.1. The package Statemachine is the code described in Section 2.2. The last package imported, as mentioned, is os. os lets the user interface with the operating system. The documentation can be found here: os.

### 2.3.3 bcolors

This class allows for text color to be set by adding attributes of this class to strings. This code is an adaptation of the code found at the following [source](#). An example of the code being used can be found in figure 3.

```
print(bcolors.FAIL+“Oh god the computer is on fire”+bcolors.RESET)
>>>Oh god the computer is on fire
```

Figure 3: Example use of bcolors

Note to the reader, if bcolors.RESET isn’t called the code will remain in the last color you set, even after the code is exited.

### 2.3.4 IO\_Devices

The IO\_Devices class was build to hold the structure of an input or output device as one object. The object has 5 attributes: name, state, pin, ras\_pin, and IO\_type.

- name - what the pin corresponds to, see Table 2

- state - whether the pin is high or low
- pin - the number of the pin according to Table 2
- ras\_pin - number of pin on the raspberry pi
- IO\_type - whether the pin is an input or output

The initialization<sup>4</sup> of IO\_Devices also has some logic for choosing whether to set the GPIO pin to be read or write.

The first method (other than init) is the setHigh method. This method sets output pins high. It first checks to see if the the pin is an input or output pin and then if it is an output pin then it checks if it is already set high. If the pin is already set high then it prints that it is maintain high, if it isn't already set high it prints that it is setting the pin high and then sets the state of the pin to high. If the pin is an input it does not set it high and prints a warning that you are attempting to set a read pin. This could more optimally be forced by creating an IO\_Device parent class with input and output children classes where the input class would not contain the setHigh and setLow methods, see Section 5.9.

Next is the setLow method. This method serves the exact same purpose as the setHigh method but sets the pin low. These two methods could technically be done in one method taking an argument for setting high vs low, however, this was decided against to improve readability.

### 2.3.5 Tasks

The tasks are the main functions which the machine communicates to the state machine. Potential changes to this organization are discussed in Section 5.4. The general idea is that the tasks contain simple logic conditions which result in setting pins using the setHigh/Low methods and then the task returns a three-tuple as discussed in Section 2.2.3. To see the logic for each task see Fig 4.

---

<sup>4</sup>For more information on classes and initialization click the word initialization



### 2.3.6 Pin definitions

### 2.3.7 Function

## 3 Hardware

### 3.1 Raspberry pi

### 3.2 Deposition machine

#### 3.2.1 PLC

This is a direct copy of what was used to run the PLC, and what the PiBrain was adapted from. **Disclaimer:** This table is not fully understood<sup>5</sup>.

Input 0 CH	IN	Terminal #	Input device	Output 5 CH	Out	Terminal #	Output device
0 PB3	0000	A0	START (PB3)	0	0500	A10	AUTO
1 PB4	0001	B0	STOP (PB4)	1	0501	B10	MANUAL
2	0002	A1	CROSSOVER	2	0502	A11	VENT
3 S8 AUTO	0003	B1	AUTO (S8)	3	0503	B11	R VAL?
4 PB1	0004	A2	ON/RESET	4	0504	A12	-NONE-
5 S8 Manual	0005	B2	MANUAL	5	0505	B12	CRYO ROUGH
6 S1	0006	A3	VENT	6	0506	A13	CRYO PURGE
7 S2	0007	B3	ROUGH (S2)	7	0507	B13	HI VAC?
8 S3	0008	B4	HIGH-VAC VALVE (S3)	8	0508	B14	WATER LOCK
9 S4	0009	A5	CRYO ROUGH (S4)	9	0509	A15	ROUGH PUMP
10 (S5) unmarked	0010	B5	CRYO PURGE	10	0510	B15	-NONE-
11	0011	A6	VACUUM IN	11	0511	B16	-NONE-
12 S6	0012	B6	ROUGH SW (S6)				
13	0013	A7	WATER LOCK				
14 PB5	0014	B7	VENT-AUTO				
15	0015	A8	-NONE-				

Table 2: Return statement logic

---

<sup>5</sup>That is not a joke

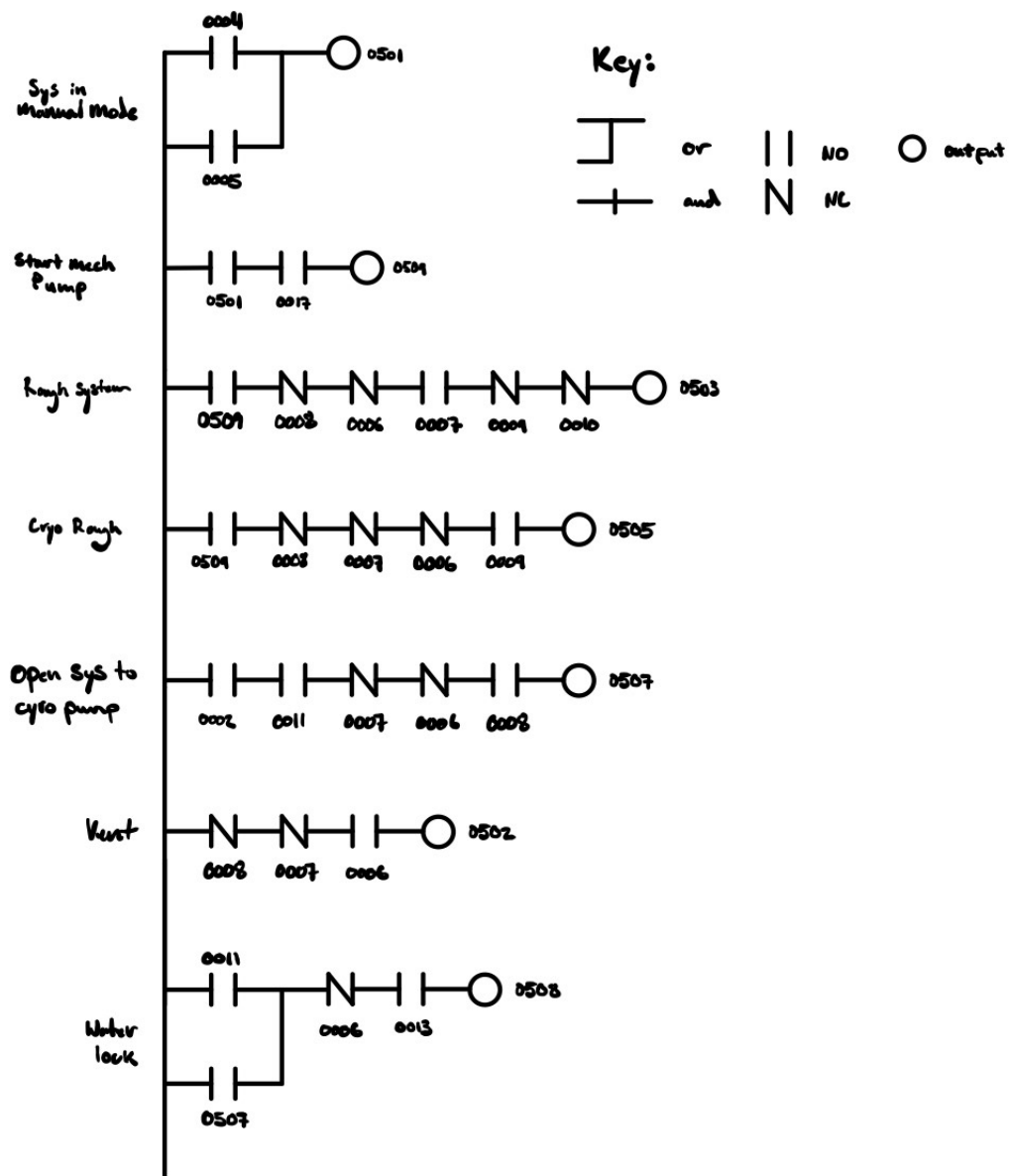


Figure 4: Ladder logic for the PLC

## **4 Installation**

## **5 Future work**

### **5.1 Combination of bash scripts with python**

### **5.2 write\_to\_log rework**

### **5.3 Idle exit**

### **5.4 Main file split**

### **5.5 Deposition controller pi**

### **5.6 Web interface**

### **5.7 GUI interface**

### **5.8 Automatic mode**

probably will need another pi, blah blah blah, pi network....

### **5.9 IO\_Device class change**