Chemical Synthesis Automation

by

Joshua Smith

Malik Sobodu

A Major Qualifying Project

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Bachelor of Science

in Computer Science

by

May 2022

APPROVED:

Smith, Therese

Abstract

In this project, our goal was to build upon existing work—specifically, a robot designed to automate chemical synthesis. The current setup already includes multiple instruments, controlled by the conductor software, which sends commands to the instruments over wireless. Our group was tasked with setting up a syringe pump to receive commands and perform its desired actions.

Contents

List of Figures

Chapter 1

Problem Description

Chemical synthesis is a process in which a number of chemical reactions are performed in order to obtain some product. Currently, it can be a tedious and time consuming task for chemists, and our hope is that this robot will be a solution to this.

A chemist will be able to enter simple commands into a LabView interface, and those commands will be converted to XML instructions that can be processed by a computer. Each device will have a different set of commands. Below is an example list of synthesis steps that could be performed using the syringe pump:

1. Turn on syringe pump
2. Move syringe pump to position 'x'
3. Draw liquid
4. Move syringe pump to position 'y'
5. Dispense liquid
6. Turn syringe pump off

Chapter 2

Overall Design

2.1 Current list of Instruments

In the current iteration, the device contains two syringe pumps, a stirrer/hot plate combo device, and a thermocouple. After viewing a list sent by our sponsor AbbVie, the potential final instrument list contains the following:

1. Syringe Pumps
2. Stirrer
3. Hot Plate
4. Bunsen Burner
5. Heating Mantle
6. Thermocouple
7. Conductance Sensor
8. Reactor Effluent Routing Valve

Other instruments—such devices for isolation of the products or analytical equipment—may be involved with the robot as well.

2.1.1 Syringe Pumps

Syringe pumps are used to depress the syringe at a set rate over a given period of time. Although two pumps will be used, we will consider it to be one single instrument. A Raspberry Pi unit with a HAT attached (which will provide us with two RS-232 ports, one for each pump) is used in conjunction with the pumps.

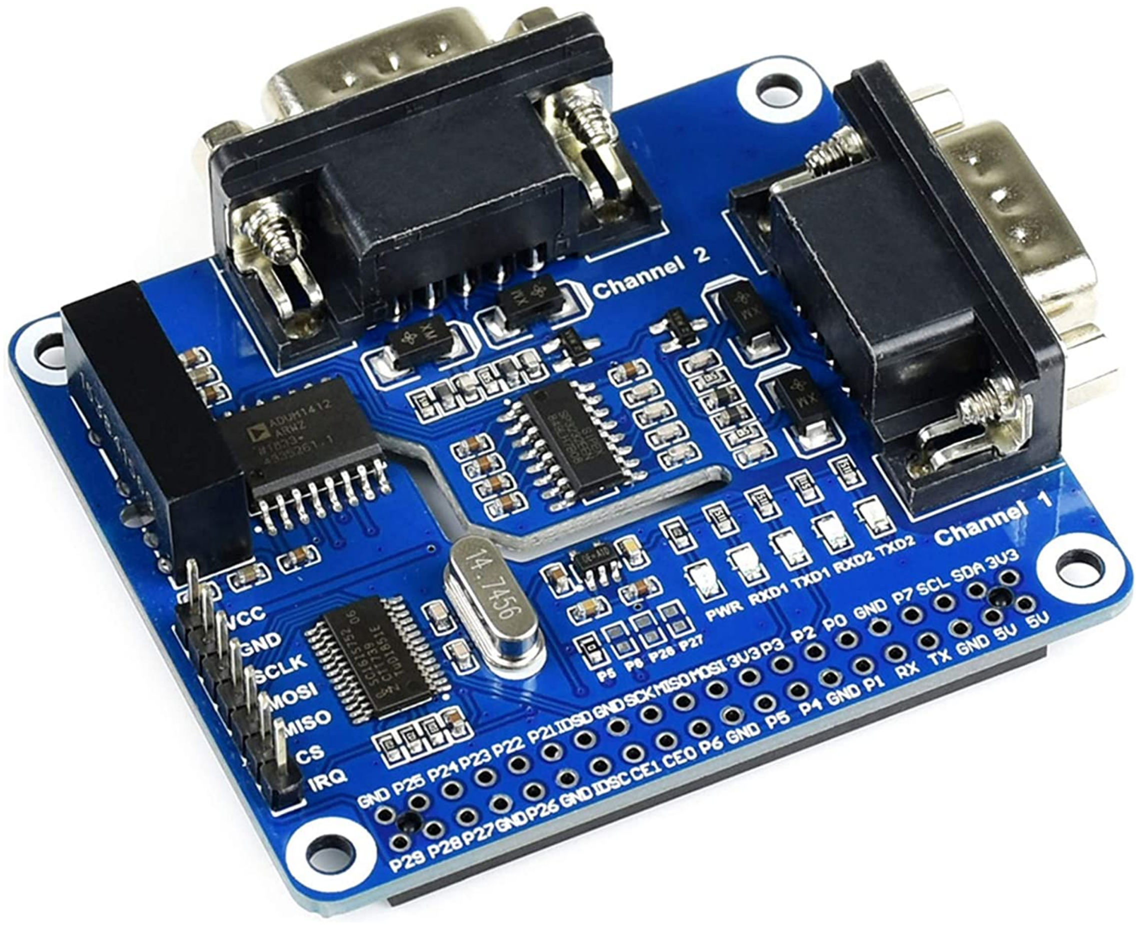

Figure 2.1: RS-232 HAT for Raspberry Pi

2.1.2 Heating

For heating purposes, we have a hot plate and a hot plate stirrer combination. It will be supported by a Raspberry Pi with a MAX3232 serial HAT, which contains a voltage level interface. There is also a simple glove-shaped heating mantle, and a Bunsen burner, which will most likely not be used often.

2.1.3 Thermocouple

The thermocouple is needed to sense the temperature during the reaction. For example, a chemist might want the robot to move to the next step in a chemical synthesis process only if the reaction reaches a certain temperature. We achieve this by using a Raspberry Pi thermocouple measurement HAT, the MCC 134.

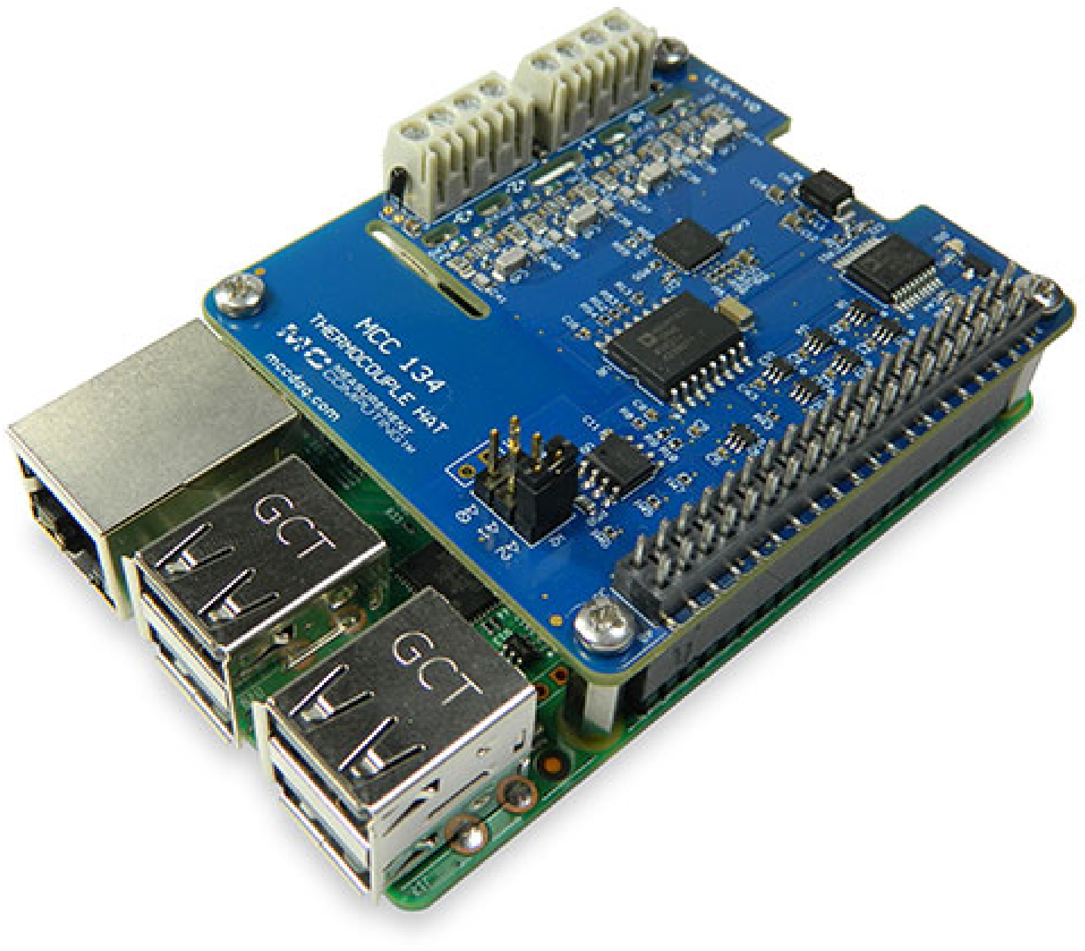

Figure 2.2: MCC 134 Thermocouple Measurement HAT for Raspberry Pi

2.1.4 Conductance Sensor and Valve

The robot will include two separate vessels for effluent, and a valve that will allow the effluent to be delivered to them. Since the valve is a solenoid, it is not suitable for a direct connection to a server computer. Instead, another method must be used to provide it with information and the necessary impedance.

A conductance sensor has also been deemed necessary, in order to determine when to switch between the two effluent vessels. The sensor will be a voltage divider, and will produce an analog voltage, which can then be converted to a digital number to be read.

2.2 RS232 Hat

HAT stands for Hardware Attached on Top. This is used to give the Raspberry Pi added features that it does not come originally with. Our specific HAT is the RS232 Expansion HAT—it is a 2 channel isolated HAT. This HAT is perfect for our MQP due to its fast communication, stability, reliability, and safety. This also makes it perfect for industrial automation. It has the standard 40 GPIO pins for a Raspberry Pi series board. The operating voltage is 3.3/5V, so we need to make sure that for whichever one we use it has a constant 3.3V or 5V supply. It includes LEDs indicating the power and transceiver status.

To begin coding with the RS232 HAT you need to set it up. To set up the software you would begin by adding these lines

```
sudo nano /boot/config.txt
dtoverlay=sc16is752-spi1,int_pin=24
sudo reboot
```

This opens the terminal and modifies the config.txt file with the commands. It then adds the line to the file and reboots the Rasberry Pi. You must then install three separate libraries—wiringpi, python2, and python3.

After the setup you can code with the RS232 HAT and Raspberry Pi freely using Python.

Chapter 3

Preparing for Serial Port Communication

As previously mentioned, our group had to start by getting the Raspberry Pi set up. The HAT is placed on top of it, aligned to its pins and it is connected to a proper keyboard and monitor.

The Raspberry Pi is a fascinating piece of equipment as it is a mobile computer. A piece of equipment like this is very delicate and must be used with extreme caution. Keeping this in mind, our first order of operation was to test the voltages on each pin. We used an oscilloscope and its probe to check how much voltage was being emitted from each pin of the Raspberry Pi/HAT combination. This was a necessary safety precaution to ensure that when we did connect it to the syringe pumps that there would be no damage done.

Our next step was to get a code editor so we could write lines and scripts to properly program the Raspberry Pi. We searched through a multitude of options such as VS Code and PyCharm but determined that the Raspberry Pi's built in text editor—Nano—would suffice for the time being. Although confusing at times, it is efficient and serves its purpose. Once that was taken care of, we began trying to figure out how to code with the Raspberry Pi. To utilize the Raspberry Pi, many commands are done through the terminal. For those unfamiliar with terminals, it is a command line system that helps you quickly take control of your operating system.

This is how we planned to make changes to code, documents, folders, software, and much more. To be proficient with using the terminal we had to familiarize ourselves with basic commands. Those commands include making new directories, changing directories, making and editing files, and more. After we did this we started to dive into the specific code. We were provided with a very helpful GitHub link that gave us a head start with coding with the Raspberry Pi.

The Raspberry Pi uses serial port communication. To begin we had to install Python and configure our Raspberry Pi for coding with it. This was done by following a ReadMe file from a Github link provided to us. The ReadMe file prompted us to install a driver that gave us access to two ports—ttySCo and ttySC1. Next, we went on to install Python 3 so we could code and run Python programs in the terminal.

3.1 x.x Config.py

To garner a better understanding of how we would utilize everything we had recently, installed we looked at the config.py file that was given, shown below:

```
#!/usr/bin/python
# -*- coding:utf-8 -*-

import serial

# dev = "/dev/ttySC0"

class config(object):
        def __init__(ser, Baudrate = 115200, dev = "/dev/ttyS0"):
```

```
print (dev)
ser.dev = dev
ser.serial = serial.Serial(ser.dev, Baudrate)

def Uart_SendByte(ser, value):
ser.serial.write(value.encode('ascii'))

def Uart_SendString(ser, value):
ser.serial.write(value.encode('ascii'))

def Uart_ReceiveByte(ser):
return ser.serial.read(1).decode("utf-8")

def Uart_ReceiveString(ser, value):
data = ser.serial.read(value)
return data.decode("utf-8")

def Uart_Set_Baudrate(ser, Baudrate):
ser.serial = serial.Serial(ser.dev, Baudrate)
```

1. Begins by importing the pySerial module. It is a serial port module that utilizes binary communications. More information can be found at the link below:

   https://pyserial.readthedocs.io/en/latest/

2. def init:

   This is a built-in function that is executed when the class is initiated (used to create a new object). This assigns values to properties that are needed when the class is initiated. The first parameter is the object name. The rest sets the parameters for each object. Ser.dev sets dev for each object to "/dev/ttyS0", ser.serial opens named port (dev) at 19200 baud rate.

3. def UartSendByte and def UartSendString:

   Both contain serial.write. This is for when you send something—in this case either a byte or a string—to your COM port. The value.encode feature encodes what you send with default "utf-8," or in our case, "ascii".

4. def UartReceiveByte and def UartReceiveString:

   This is the opposite—for when you receive a string or byte from the port.

5. def UartSetBuadrate:

   This sets the baud rate of the port— how fast the info is transferred over a communication channel. This file is where the serial port class and the necessary serial port functions are stored for later use.

3.2 x.x Main.py

We then went on to the main.py file, shown below:

```
#!/usr/bin/python
# -*- coding:utf-8 -*-
import serial
import os
import sys
import logging
```

```
logging.basicConfig(level=logging.INFO)
libdir = os.path.join(os.path.dirname(os.path.dirname(os.path.realpath(__f
if os.path.exists(libdir):
    sys.path.append(libdir)

from waveshare_2_CH_RS232_HAT import config

ser = config.config(dev = "/dev/ttySC0")
data = ''
ser.Uart_SendString('Waveshare 2-CH RS232 HAT\r\n')

try:
        while(1):
        data_t = ser.Uart_ReceiveByte()

        print("%c"%data_t),
        ser.Uart_SendByte(data_t)

except KeyboardInterrupt:
    logging.info("ctrl + c:")
        exit()
```

1. It imports config.py.
2. It then utilizes the config.py class and makes a serial port and sends it to "/dev/ttySC0".

3. Sets the variable data to a blank string that will get altered later in a try-except block.

4. Sends the string 'Waveshare 2-CH RS232 HAT' to the serial port created above.

5. In the try-except block it attempts the try and if it fails it reverts to the except statement.

6. Inside the try statement there is a repeating while loop that utilizes the blank variable and sets it to the byte received from the serial port. It then prints it and sends it back.

3.3 x.x Test.py

Test.py followed a similar structure but focused on communicating between two channels. Specifically, sending a string between two ports. When running both of these we only got the port location, and it would not iterate through the code, so more work was required to determine what the problem was. We must first solve this problem and then read messages received from a logic analyzer before moving on to controlling the syringe pump.

Chapter 4

Controlling the Syringe Pump

We know how to code the machine and have a general understanding of how the hardware is supposed to connect. RXD is pin 2, TXD is pin 3, and Pin 1 is the VDC from the MAX232. This gives the power to the syringe pump (PSD/6). The MAX232 will give enough power for two applications connected (in our case the two syringe pumps). We will communicate serially to these two pumps. Our next step is to develop the code to control the PSD/6.

4.1 Necessary Code and Commands

We will be in standard protocol and standard resolution when communicating back and forth with the PSD/6. Standard protocol has a specific way of coding that communicates to the PSD/6 and it does it with a command buffer. This command buffer is technically an array that has at minimum 6 spots in the array and each spot contains an 8-bit binary number or hex value, for example, 0x58. These 6 spots contain an STX which is a hard coded hex value, an address which is either bit 0 or 1, and a sequence number in which most are previously set, except bits 0 – 3 (contains 8 total bits). Bits 0-2 are used to control the sequence number, which is the order of the commands. This sequence number goes up to 8 and then resets to 0. This is necessary as two consecutive commands cannot have the same sequence number. The code used to cycle through these sequence numbers is shown below:

Figure 4.1: PSD/6 Precision Syringe Pump

```
int sequenceI = 30+syringePumpSeq;
char sequenceC = sequenceI+ "0";
syringePumpSeq++;
syringePumpSeq = syringePumpSeq%8;
if(syringePumpSeq<0) syringePumpSeq += 8;
standardProtocolCommand[2]=sequenceC; //seq of message
```

The variable sequnceI contains the number that will be added to bits 0-2 of the sequence number. It starts at 30 and goes up to 38. It then gets turned into a char and made into 3 bits by appending 0 to the end. The final values that would be added to the sequence number portion of the sequence data would be '300', '310', ‘320’, up to ‘380’. We will always get a singular digit number to add to 30 in sequneceI due to us using the remainder a remainder of 8%. This will always give us a number between 0 – 8 to add to 30 maintaining a loop. The next portion of the command buffer is the command set, which is all the commands we would need to control the PSD/6. As of now these commands will be hard coded in. Below are descriptions of some commands that can be used:

1. Ix:

   This is apart of the valve commands. This specific valve command moves the valve to the input position. It is set to input position before moving the absolute position up to draw in liquid.

2. R:

   This is part of the execute commands. It is placed at the end of the command buffer and tells the PSD/6 to execute the commands in the buffer

3. Sx:

   This is apart of the motor command controls which set up the syringe pump motor. It sets the speed ranging from 1 to 40.

After the command set, the final two pieces of the command buffer are the ETX, which is a hard coded value that symbolizes the end, and STX which symbolizes the start. After the ETX there is the checksum. Checksum adds up the bits in each column and if it is sets it to 0 if even, and 1 if odd. This needs to match the checksum that the PSD/6 gives as a response. It is used to make sure the command that we want to send is the command received by the PSD/6. It takes in a list that contains the command buffer lines. Each spot in the command buffer array has an

8-bit binary number and or hex number in it. It takes all the bits in column 0 of the array and adds them together.

4.2 Testing and Results

Unfortunately, our group was unable to fully configure the syringe pump to be able to receive commands and perform chemical synthesis steps. Communicating with the serial port of the Raspberry Pi and its HAT proved to be very difficult.

Chapter 5

Conclusion

Overall, our project went well. We learned a lot about working with Raspberry Pis and other devices, which will surely help us in future work. We were not able to completely achieve our goal, but we hope that future groups will be able to take our contributions and use them to make the syringe pump functional.