ASPIRE Control System Software Documentation

Hayden Klassen

August 30, 2024

This document presents a detailed description of the initial build of the TRIUMF ASPIRE control system software, and an explanation on how it was constructed. Recommendations for operating and updating the control system are outlined, some implicitly. This document is not structured as a concise operation manual for the control system, and some knowledge of EPICS software tools is assumed.

Contents

1 Introduction

In order to facilitate the control of ASPIRE devices: electrical instruments and mechanisms composing the physical ASPIRE experimental station, a control system is employed. Fundamentally, the control system is a network involving an operator computer, IOC computers, and the ASPIRE devices to be controlled. The operator computer is a user-accessible computer that provides the control system operator with a graphical interface for inputting commands that execute actions through the IOC computers and onto ASPIRE devices. The IOC (input/output controller) computers receive commands from and send information to the operator computer, while translating digital information to and from electrical signals that execute actions on ASPIRE devices.

2 Operator Computer

The operator computer is the device that the human operator uses to perform actions upon ASPIRE devices. The current operator computer natively runs on Microsoft Windows 11 OS, but the control system is interfaced through a Linux subsystem: Windows Subsystem Linux (WSL). On WSL, EPICS is installed, in addition to the extension EDM for the control system GUI, and the extension StripTool for plotting PVs, in addition to other dependencies.

2.1 Linux Subsystem

On the Windows operator computer, WSL 1 is installed along with the Linux distribution Ubuntu LTS version 18.04.6. This Ubuntu version is installed from the Microsoft Store. The installation process of an Ubuntu subsystem with WSL is outlined in section 4.1 of (Henders et al. 2024), or found elsewhere on the internet. WSL creates a Linux kernel virtualized through Windows Hyper-V, which allows for the use and control of Linux-developed software and tools like EPICS and its extensions on a native Windows OS computer. Almost all interactions with the control system are directly or indirectly performed through this Ubuntu subsystem.

2.2 EPICS Installation

EPICS software tools structures and communicates the data of the control system between computers, and therefore must be installed on the operator computer within the Ubuntu subsystem. EPICS version

3.15 will be used in the control system.

The source code for EPICS 3.15 must be downloaded, along with a plethora of contingent libraries, and then be compiled. The full installation process is outlined by section 4.13 in (Henders et al. 2024). It is strongly recommended to not directly copy and paste the provided bash commands into the Ubuntu subsystem terminal, as the hyphen and quotation Unicode characters are incorrect. Instead, the commands should be completely re-typed. As an overview, the installation process involves setting the system environment variables in the bashrc files so that other software and tools are aware of the location of EPICS files and settings, then downloading the EPICS-BASE source code from https://github.com/epics-base/epics-base, installing required library packages, then compiling the source code using GNU make.

Common compiling issues include: missing libraries, fixed by reading error message and searching for the missing library name, then using the "sudo apt-get install [package name]" command to install it; error for being unable to locate EPICS directory or identify computer architecture, fixed by correcting the environment variable entries for "PATH" and "EPICS_BASE" in the bashrc files to point to the correct file directory of the EPICS installation. The EPICS installation can be verified by running the "softIoc -x test" command in the Linux terminal. If the command fails, but running the softIoc file located at "/epics-base/bin/linux-x86_64/softIoc" is successful, then there is an issue with the environment variables in the bashrc files; otherwise, EPICS did not properly compile. If the empty IOC server starts, EPICS has been installed.

2.3 EDM and StripTool

The EPICS extensions EDM and StripTool are also installed, along with the dependent Windows software Xming. EDM is used to create a GUI that interfaces with EPICS PVs, allowing for the operator to manage the ASPIRE control system. StripTool adds the functionality of graphically plotting a PV value over time.

2.3.1 EDM Installation

EDM must be compiled from its source code and correctly integrated with the main EPICS installation as an extension. EDM's source code should be downloaded from https://github.com/epicsdeb/edm and to the EPICS directory at "/opt/epics/extensions/src/". As an overview, EDM system environment variables should be added to bashrc, and additional libraries like Motif/OpenMotif, zlib, libpng, and libgif must be installed. EPICS configuration settings must be changed to alert it of the EDM extension. In addition to installing EDM on the Ubuntu subsystem, the software Xming and Xlaunch are required to be installed natively on the Windows OS in order to display the graphical windows from EDM. Detailed installation instructions are found at the given EDM GitHub repository, or through section 4.13-4.23 in (Henders et al. 2024). On Windows, the newly-installed Xlaunch program should be executed with the default settings but with the "No Access Control" setting enabled, thereby initializing the Xlaunch server. Any time EDM should be used, an Xlaunch server must be activated. Once EDM is successfully compiled, and an Xlaunch server has been created, EDM can be started by running the command "edm" on the Ubuntu subsystem. If a small access window opens, then EDM has been successfully installed.

2.3.2 StripTool Installation

StripTool, like EDM, is an EPICS extension that must be compiled from its source code and integrated with the EPICS installation. StripTool's source code should be downloaded from https://github.com/epics-extensions/StripTool to the EPICS extension source directory "/opt/epics/extensions/src/" by running the command "git clone https://github.com/epics-extensions/StripTool.git" in that file directory. StripTool dependencies, including necessary libraries, can be found at https://epics.anl.gov/extensions/StripTool/ and https://epics.anl.gov/extensions/medm/. All of these requirements should already be satisfied by following the installation procedure for EPICS and EDM, with the exception of the library libxpm-dev, which is installed using the command "sudo apt-get install libxpm-dev". Then, StripTool is compiled by running the "make" command in the installation directory "/opt/epics/extensions/src/StripTool/". If any compilation errors are encountered, read the error message for a clue about the error source. A likely source of error could be a missing library, whose name will be stated in the error message. Once compiled, the StripTool installation can be verified by running the command "StripTool" in the terminal. If not recognized, then either StripTool was not successfully compiled, the PATH system environment variable modified in bashrc does not point to the EPICS extensions directory, or StripTool was not installed in the correct directory. If the command opens a "StripTool Controls" window, then StripTool has been successfully installed.

2.3.3 Development with EDM

EDM offers a simple, graphical, development environment for control system GUIs. After the Xlaunch program has been launched on the Windows OS, the EDM access window is opened by running the "edm" command in the Ubuntu terminal. A new, customizable, EDM window is created by clicking “File” and then “New”. This new EDM display can be saved as an .edl file.

Basic Functions

Some essential mouse and keyboard controls for development in EDM are:

- Left-click, drag the mouse, and release to access the widget menu. It allows for the placing of graphical widgets that the user can view, monitor PVs with, and control PVs with.
- Middle-mouse click to open a menu that allows for the switching between execution and edit modes of the GUI, saving the design to file, configuring display properties, among other options.
- Once a placed widget has been selected by left-clicking, left-clicking again opens the configuration menu for the widget.
- After left-click selecting a widget, holding down while dragging the mouse will change the position of the widget, and dragging the corners will re-size the widget.
- Holding down shift and left-clicking two widgets will select more than one widget at a time. Multiple selected widgets can be grouped as one object using the “group” option in the middle-click menu.
- While a widget is selected, pressing the arrow keys will move the widget with fine control.
- If a widget is blocking access to another widget, tapping the “d” key while a widget is selected will move it backwards and remove its click-priority, and tapping the “u” key will move it forwards.

Within the middle-click-accessed “Display Properties” configuration menu, various display settings can be altered. The “Height” and “Width” options change the sizing of the display window, the background colour of the display window can be changed through the “Display Bg” option, and the positional grid on which the widgets snap to is changed with the grid options.

Widgets and Design

Once a widget has been selected from the widget menu, a configuration menu will appear, where position, size, color, and more can be customized for the appearance of widgets. PV monitoring widgets have a “PV” or “Readback PV” option which, when a PV name is entered, will read out the PV’s value. PV control widgets have a “Control PV” option which, when a PV name is entered, will allow for inputs into that widget to change the value of the given PV. Each widget type has its own unique configuration settings.

The most fundamental monitoring and control widgets follow. For visual structure, the "Rectangle" widget is used to segregate the display into a grid of elements, and the "Static Text" widget is used to write the header texts. For interaction with PVs, the "Choice Button" widget is used as the main 2-choice button to switch binary PVs between 0 and 1, and the "Text Monitor" widget is used to display the value of a PV as text. For further functionality, the "Related Display" widget acts as both a PV label for an adjacent widget like a text monitor, while also serving as a button that opens a new display window when clicked. As the new display window is another .edl file, the file must be created in-advance, and then be specified as the target file in the "File" option of the "Related Display" widget. Terminal commands, like the command "StripTool" which opens a StripTool window, can be executed at the press of a button using the "Shell Command" widget. For more information on the widgets used in the ASPIRE EDM control system display, double-click to access the widget configuration menu while in EDM edit-mode to view the prescribed settings, or edit them experimentally to explore possible functions. Some documentation can also be found at (Sinclair 2007), and the documentation page built-into EDM, which is accessed through the initial EDM access window by clicking "Help".

Some widgets can be made to change color or visibility based on a PV information. To make a widget's color change based on a PV's alarm state, select the "alarm sensitive" box under the widget's color options within the configuration menu, and then enter the PV that the widget should respond to in the "Color PV" box. This is especially useful in making indicator lights, like the circular indicator used to convey if RPi Cameras are on or off. If the PV's alarm state is "NO_ALARM", then the widget will show the standard configured color, and if the PV's state is "MAJOR_ALARM", then it will show the major alarm red color where the "alarm sensitive" box was ticked. For the respective colors and behaviors of all alarms in EDM, see section 3 of (Sinclair 2007). Toggling visibility of a widget using the value of an analog PV is possible by entering the PV name in "Visibility PV" and then configuring the visibility logic below.

Using these tools, the ASPIRE EDM control system GUI was created in EDM. Further documentation on EDM is found at (Sinclair 2007).

2.4 Starting the ASPIRE EDM Display

The operator of the control system must be aware of how to start the ASPIRE EDM display. There are two ways in which the ASPIRE EDM Display can be started. The recommended manner is by executing the "Launch EDM.bat" file on the operator computer Windows OS, which automates the commands involved in the second method: manually starting Xlaunch and then opening the ASPIRE EDM GUI.

To start the ASPIRE EDM Display manually, first confirm that an Xlaunch server is running, and if not, start the Xlaunch app on Windows OS with default settings but with “No Access Control” enabled. Then, open the Ubuntu subsystem terminal, and navigate to the “/aspire_GUI/” folder located in the user directory. Followingly, execute the terminal command “edm -x -noedit main_menu.edl”. This command opens the main ASPIRE EDM window, in the state of executed mode (-x) and no edits mode (-noedit). The main_menu.edl display is the nexus for all other display windows through the graphical “Related Display” widgets, but any EDM display window can also be opened from the command-line using the “edm [file name].edl” command. Thus, the ASPIRE EDM control system display can be started manually by first starting Xlaunch and then opening the EDM ASPIRE main menu window with an edm command.

The steps involved in starting the ASPIRE EDM display manually have been condensed into a Windows batch file. Executing the “Launch EDM.bat” automatically starts the Xlaunch server and then executes the same edm command for the Ubuntu subsystem user “SERVER-ROOT”. The exact commands the batch file runs are the following:

```
start cmd.exe /c "C:\Program Files (x86)\Xming\Xming.exe" -multiwindow
sleep 2
wsl.exe bash -ic "cd /home/SERVER-ROOT/aspire_GUI; edm -x -noedit main_menu.edl"
```

The first line starts a separate Windows command terminal and runs Xming.exe to start Xlaunch with the multi-windowed option enabled. Then, after a 2 second delay, an interactive (-ic) Ubuntu WSL shell is opened, and the previously seen edm command to start the main ASPIRE GUI menu is started from within the “aspire_GUI” folder. If this script is to launch an ASPIRE EDM display on another computer, the given WSL username in the script must be changed from “SERVER-ROOT” to the currently used username, and the location of the Xming folder must be verified to be in C:\Program Files (x86). In this manner, the ASPIRE EDM Display can be started automatically.

3 Raspberry Pi IOCs

The several Raspberry Pi (RPi) that compose the ASPIRE control system act as IOCs. They host EPICS PVs, sending their data to and receiving it from other RPi IOCs and the operator computer. They then send electrical signals to ASPIRE devices to perform actions, or receive electrical signals from ASPIRE devices and update PV values accordingly. These RPi IOCs communicate with the other computers on a wired network-switch LAN. These RPi must have their Raspberry Pi OS set up for running EPICS, and then configured to facilitate a consistent and secure connection with other computers on the LAN. PV data sent to the RPi is read and written to using continuously running python scripts, which also control the GPIO pins on each RPi, thereby managing the behavior of ASPIRE devices. In these ways, do the RPi fulfill the role of IOC in the ASPIRE control system.

3.1 Raspberry Pi Flashing

A microSD card must be flashed with a version of Raspberry Pi OS for use in a Raspberry Pi. This is best done by flashing the pre-imaged ASPIRE RPi file, or by flashing a fresh Raspberry Pi OS.

3.1.1 Pre-Imaged ASPIRE RPi

An image has already been taken of a set-up ASPIRE RPi IOC, which can be flashed on a microSD card, for use in a Raspberry Pi 3, to skip the following RPi setup steps explained in section 3.2. Though, a novel static IP network address must still be set on the RPi. Note that this image file only exists for the Raspberry Pi 3. This flashable image file is found under the name "ASPIRE_IOC.img". The image is flashed onto a microSD using the Raspberry Pi Imager tool (https://www.raspberrypi.com/software/). With this software, insert a microSD card into the flashing computer, select the Raspberry Pi 3 device type, choose "Use Custom" under the "Operating System" option and select the ASPIRE RPi IOC image file, and then select the microSD card under the "Storage" option. When prompted to customize OS settings, select the "edit settings" option. Under the "General" header, set the hostname to a suitable name for the device, such as "rpi008", then set the username to "aspire" and the password to "root". Do not configure wireless LAN. Under the "Services" header, enable SSH and select "Use password authentication". After saving and clicking "yes" to apply the OS customization settings, the microSD card will then be flashed, and can be inserted into the RPi when complete.

3.1.2 New RPi Image

A fresh version of Raspberry Pi OS can also be flashed to a microSD card using the Raspberry Pi Imager tool. After inserting a microSD card into the flashing computer, select the correct Raspberry Pi device, then select Raspberry Pi OS (32-bit) as the operating system. Note that EPICS is only supported for 32-

bit Raspberry Pi ports of Debian Linux. Select the microSD card as the storage device and proceed. When prompted to customize OS settings, select the "edit settings" option. Then, make the same customizations outlined in section 3.1.1, flash to the microSD card, and insert into the RPi when complete.

3.2 Raspberry Pi Setup

System settings on a freshly-flashed RPi must be changed to make it a suitable IOC. If the ASPIRE RPi image was flashed, then the only procedure that must be followed in this section is the configuration of novel static IP address. The following guide assumes access to the graphical desktop environment of the Raspberry Pi OS, accessed with an HDMI connection and USB mouse and keyboard.

3.2.1 System Configuration

Various small system settings should be changed on the RPi. Some important settings can be changed by navigating to the Raspberry Pi Configuration menu, accessed by clicking the Raspberry Pi icon drop-down menu at the top left, clicking "Preferences", then "Raspberry Pi Configuration". Once there, enable "Auto Login" under "System", disable "Screen Blanking" under "Display", and enable every option under "Interfaces". Then, save the settings and reboot. On the main desktop display, Bluetooth and audio can also be disabled as they are not necessary.

3.2.2 Network Configuration

A network connection must be established on every RPi so that they can communicate with other RPi IOCs and the operator computer. Namely, each RPi must been connected by Ethernet to the ASPIRE network-switch LAN, and optionally a Wi-Fi connection can be established.

E-LAN Connection

Every RPi must be connected to the network-switch E-LAN so that they have a secure and private communication network with each other and the operator computer. Begin by connecting the RPi to the network-switch by Ethernet cable. Then, the RPi should recognize the LAN network but not successfully connect. To establish a proper Ethernet LAN connection, first navigate to the network configuration settings by first clicking on the network icon by the top right of the desktop display, then disable Wi-Fi by clicking "Turn Off Wireless LAN". While in this network menu, hover over "Advanced Options" and click

"Edit Connections". Once in the edit connections page, select and click configure for the Ethernet network connection option. A detailed menu will open containing all configuration options for this Ethernet LAN network. Navigate to the "IPv4 Settings" page, and switch the "Method" option to "Link-Local Only", which allows for the network-switch to assign an address to the RPi. Then save, and the Ethernet connection should successfully establish.

To make this Ethernet LAN connection static, so that the IP address does not change, more specific configurations to the network must be made. In the network settings menu for the Ethernet connection, and under the "IPv4 Settings" page, change the negotiation method from "Link-Local Only" to "Manual". Open a terminal window and execute the command "ifconfig". Copy the IPv4 address output in the terminal window under the section "eth0" and after the word "inet", and paste this IP address into the "Address" text box. Then, copy the netmask address that follows after the inet address given in the terminal, and paste it into the "Netmask" text box in the IPv4 Settings page. Then, save the new network settings, and the connection should re-establish itself. Reboot the RPi and verify that the statically defined LAN connection is successful. Note this IP address down, preferably in the IPADDRS.txt file on the operator computer where all ASPIRE RPi IOC IP addresses are stored.

W-LAN Connection

A Wi-Fi, or W-LAN, connection can also be established in addition to the E-LAN network. Note that this should only be done when the RPi requires access to Wi-Fi, as connecting opens vulnerabilities in the ASPIRE IOC network, where computers not associated with ASPIRE can communicate with the RPi and potentially interfere with the control system. Start with enabling Wi-Fi by clicking the network icon at the top right of the desktop display, and then clicking "Turn On Wireless LAN". Followingly, select the desired Wi-Fi network TRIUMF Secure. An authentication screen will appear, where credentials must be provided to access the Wi-Fi network. Change the "Authentication" option from "Tunneled TLS" to "PEAP", write one's TRIDENT username in the "Anonymous identity" text box, select the "No CA certificate is required" check box, then enter one's TRIDENT username and password in the respective "Username" and "Password" text boxes. After proceeding, the Wi-Fi connection should establish itself.

The previously established Wi-Fi connection is dynamic, and should be defined instead as static to keep the same IP address. Restart the RPi and then navigate back to the network connections configuration menu, where the "TRIUMF Secure" Wi-Fi network will be selected and then configured. Select the "IPv4

Settings" page on the TRIUMF Secure connection configuration menu, and change the connection method to "Manual". Then, open a terminal window and execute the command "ifconfig". Copy the output inet address and netmask address under the "W-LAN0" section, and paste them into their respective text boxes in the IPv4 settings page. Then, execute the command "ip r", and copy the gateway address output after the phrase "default via" into the "Gateway" text box in the IPv4 settings page. Lastly, execute the command "cat /etc/resolv.conf", and copy the nameserver address(es) into the IPv4 settings text box called "Additional DNS servers". If there is more than one nameserver address given, separate them with a comma. Click save, and the connection should re-establish itself. Reboot the RPi and verify that the statically defined W-LAN connection is successful. Note this IP address down, preferably in the IPADDRS.txt file on the operator computer where all ASPIRE RPi IOC IP addresses are stored.

Through the aforementioned steps, an Ethernet LAN (E-LAN) connection is formed on each RPi, with the optional addition of a Wi-Fi (W-LAN) connection. This allows for communication between RPi IOCs and the operator computer.

3.2.3 EPICS Installation

EPICS is installed on the RPi to enable EPICS IOC functionality, allowing for the creation of an EPICS PV database and and the sharing of PV information. In a similar process to installing EPICS on the Ubuntu subsystem in section 2.2.1, first the EPICS-BASE source code is downloaded from https://github.com/epics-base/epics-base, then environment variables are set in bashrc, and once required libraries are installed, EPICS is compiled. Detailed installation instructions are found in section 3.1 of (Henders et al. 2024). The EPICS installation can be tested by running the command "softIoc -x test".

3.3 RPi IOC Program Structure

There are several separate files and scripts that compose the IOC on each RPi. Namely, a crontab calls a startup shell script, which then calls a python script to generate the EPICS PV database file, and then creates an EPICS IOC server using that database, and finally the startup script executes the IO python scripts which run continuously to handle the logical and electronic IO functionality of the IOC.

3.3.1 Boot-up Cron Job

On boot-up of each RPi, a cron job is executed which automatically triggers the startup of the IOC by executing the bash script startup.sh located in the "/home/aspire/rpi###_IOC/" directory. A new cron job is created by first executing the terminal command "crontab -e", offered by the package cron. Once the crontab file opens, below the comments, the addition of the line "@reboot /home-/aspire/rpi###_IOC/startup.sh" will cause the startup.sh file to execute on boot of the RPi if "rpi###_IOC" is the name of the IOC folder transferred to the RPi. If none of the RPi IOC services start on reboot, it is likely because the startup.sh file is missing execution permission. The execution permission can be added by running the command "chmod +x startup.sh" in the appropriate directory.

3.3.2 Startup Shell Script

The startup.sh script that is executed by the boot-up cron job is responsible for calling all of the other necessary scripts that perform IOC functions, including the generation of the EPICS PV database file from a corresponding python script, starting the EPICS IOC server using it, and then executing the remaining python scripts which perform IO functions. In order to have multiple commands and scripts run concurrently, the package "screen" is used to execute each command in a separate 'detached' background terminal. The following text is the startup.sh script for rpi001:

```
#!/bin/bash
echo "STARTING EPICS SERVER"
pkill screen
sleep 1
cd /home/aspire/rpi001_IOC
screen -d -m -S create_db.py python create_db.py
sleep 2
screen -d -m -S epics.db /opt/epics/epics-base/bin/linux-arm/softIoc -d epics.db
screen -d -m -S valve_control.py python valve_control.py
screen -list
echo "SERVER STARTED"
```

In this script, the first functional command, "pkill screen", kills the process of any previously existing screen instances. Then, after a 1 second delay, the directory is changed into the IOC folder and a detached terminal window named "create_db.py" runs the command "python create_db.py" which executes the create_db.py file that generates the EPICS PV database file. In the screen command, the options "-d -m" make the terminal window detached and hidden in the background, and the option "-S" names the terminal session the proceeding word. After initiating the generation of the EPICS PV database and a 2 second delay, startup.sh creates the EPICS IOC server using the epics.db database by running the command "/opt/epics/epics-base/bin/linux-arm/softIoc -d epics.db" through screen. This command is analogous to "softIoc -d epics.db", but must specify the full file path of the softIoc command file because the PATH system environment variable which makes the shell aware of the location of command files is not loaded. Hence, the EPICS IOC server has been started, with all intended PVs loaded. Afterwards, the valve_control.py python script is executed, which runs continuously to handle all IO functions. On different RPi IOCs, other IO python scripts will run in its place. Lastly, the screen sessions will be listed. If a detached screen terminal is desired to be re-attached and brought to the foreground, such as for diagnostic purposes, the command "screen -r [session name]" should be run. Thus, all scripts necessary to the function of the IOC have been called by the startup.sh script.

3.3.3 EPICS Database Generation and IOC Server

One of the python scripts executed by the startup.sh script is create_db.py. This python script generates the EPICS database file responsible for creating PVs, based on the addition of PV information to the script.

The generated EPICS database file is composed of records which will each create a PV. Each record has a type to represent the datatype of the process variable to be created. For instance, the record type "ai" means a record of the type "analog input", where such a PV will receive and store analog values as inputs. Another important record type is "bi", meaning a binary input record which receives and stores binary values as inputs. Each record must also be defined with fields to specify further information about the record, such as the field "INP", which is used to specify the input or initial value of the PV or what other record it receives its initial value from. Other important fields include "SCAN", which is used to specify how often the PV should update its value and alarm states, the field "EGU", which specifies engineering units for the PV, and the fields "ZSV" and "OSV", which respectively denote the alarm severity of a binary PV when the value is 0, and the alarm severity when it is 1. An excerpt of two records from the rpi005 database file follows:

```
record(ai,ASP1:CGB:RDVOLT){
```

```
field(INP,'0')
field(EGU,'V')
field(SCAN,'1 second')
}
record(bi,ASP1:CGB:STATON){
field(INP,'0')
field(ZSV,'MAJOR')
field(OSV,'NO_ALARM')
field(SCAN,'1 second')
}
```

Detailed information on records and fields can be found in the EPICS Record Reference Manual https://epics.anl.gov/EpicsDocumentation/AppDevManuals/RecordRef/Recordref-1.html.

The python script create_db.py, called by startup.sh, generates the described EPICS database file. It parses the PV information that is embedded into its own script, and translate it into the seen-above syntax of an EPICS database record, then writes it to the epics.db file. This create_db.py script is the file that is modified by the operator to add, change, or remove PVs on the particular RPi IOC. It is purposefully written in a more concise and readable syntax than an EPICS database file. A modified extract from the rpi005 create_db.py file is as follows:

```
import os
def create_database():
    # create and initialize writing in epics.db file
    if os.path.exists("epics.db"):
        os.remove("epics.db")
    datab = open("epics.db", "w")

    # function to parse given record and field information
    def create_record(record_name: str, record_type: str, **fields: str):
        record_entry = 'record(' + record_type + ',' + record_name + '){\n'
                for field_type, field_value in fields.items():
                        record_entry += 'field(' + field_type + ',"' + field_value + '")\n'
                record_entry += '}\n'
                datab.write(record_entry)

    ####################################
```

```
17      # WRITE PV RECORDS HERE TO BE ADDED TO DATABASE
18      # add in the form create_record("PV NAME", "PV TYPE", FIELDTYPE1="FIELD VALUE 1", ...)
19
20      # pressure gauge PVs
21      gauge_list = ["ASP1:CGB","ASP1:CGA","ASP1:CG1"]
22      for gauge in gauge_list:
23          create_record(gauge + ":RDVAC", "ai", INP="0", EGU="T", SCAN="1 second")
24          create_record(gauge + ":RDVOLT", "ai", INP="0", EGU="V", SCAN="1 second")
25          create_record(gauge + ":STATON", "bi", INP="0", ZSV="MAJOR", OSV="NO_ALARM",
26                                                      SCAN="1 second")
27
28      datab.close()
29  create_database()
```

The main function create_database() is called, which will generate the EPICS database file. Lines 4-6 handle the deleting an old epics.db file if it exists, and then creating a new epics.db file with the datab object to handle writing to it. Then, the create_record function is defined, which will be called each time a new PV record is desired to be written to the epics.db file. It essentially parses the given inputs into a string of the multi-lined syntax of an EPICS database record, and then writes this string to the database file. This function requires the record name, record type, and an arbitrary number of field type parameters to be given when called. The remaining lines call the create_record function to write a PV record, such as the for-loop used to create 3 PV records for each one of the convectron gauges. Then, the file writing session is closed with “datab.closed()” and the create_db.py script ends. Thus, the create_db.py python script has generated the necessary epics.db EPICS database file, which contains all the records used to generate the PVs on each RPi IOC server at startup when the “softIoc -d epics.db” command is run.

3.3.4 Python IO Scripts

The final python scripts that startup.sh executes are the IO scripts. They change the voltage state of their RPi GPIO pins based on received PV information, and change the value of a PV based on received GPIO pin voltages. This allows them to interact with ASPIRE devices. For instance, the IO script on rpi001 actuates the valves on ASPIRE, and the IO script on rpi005 reads voltage outputs from vacuum gauges to determine pressure values. The IO scripts can also perform special software-ended functions, as in streaming camera footage like rpi004, and querying external TRIUMF ISAC PV values as does rpi003.

In order to make calls to EPICS from a python script, the python module “pyepics” must be installed. If not already present, it is installed with the terminal command “pip install pyepics”. Pyepics streamlines access to EPICS Channel Access (CA) for PVs, offering abstractions like the creation of a PV object, that allows for easy manipulation of PV information, and useful functionalities like callbacks.

Valve Control Script

The valve control python script on rpi001, named valve_control.py, has the purpose of actuating the valves on ASPIRE and turning on the backing pump. It controls these ASPIRE devices by sending a voltage signal from its RPi GPIO pins to a relay connected with the valves, based on the value of the respective PV for each ASPIRE device. A modified version of the script is the following:

```
import RPi.GPIO as GPIO
from epics import PV

# this python file controls the valves and backing pump on aspire.
def main():

    #############################################
    # WRITE OUTPUT PVs HERE. Nested arrays must be of the form [PV("PV NAME"), GPIO PIN #]
    out_pv_list = [[PV("ASP1:RV1"),21],[PV("ASP1:PV1"),20],[PV("ASP1:VVA"),16],
        [PV("ASP1:BV1"),12],[PV("ASP1:BP1:STATON"),26],[PV("ASP1:SPARE19"),19],
        [PV("ASP1:SPARE13"),13],[PV("ASP1:SPARE6"),6]]
    #############################################

    # gpio pin initialization.
    GPIO.setmode(GPIO.BCM)
    for pv in out_pv_list:
        GPIO.setup(pv[1],GPIO.OUT)

    # function to commit change of pv state into gpio pin state
    def make_change(pin, pv_value):
        # note reversed logic here. Change if relay circuit fixed.
        if pv_value == 0:
            GPIO.output(pin,GPIO.HIGH)
        else:
            GPIO.output(pin,GPIO.LOW)
```

```
26
27      # main running loop. Handles pv-pin commands
28      while True:
29          for pv in out_pv_list:
30              pv_value = pv[0].value
31              make_change(pv[1], pv_value)
32  main()
```

In the main function on line 9, it is seen that the array out_pv_list is declared with each sub-array's elements being a pyepics PV object to represent a valve or backing pump ASPIRE device, and its corresponding GPIO pin for controlling that device. This array is used to pair PV information with GPIO pin states. Lines 14-16 initialize the pin naming convention and sets all GPIO pins in the out_pv_list array as output pins. Lines 19-24 define the function make_change, which will be called in the main run-loop to commit a change of a binary PV's value to the binary state of the respective GPIO pin. Followingly, lines 30-33 contain this main run-loop, which continuously handles getting the value of each PV in out_pv_list, and then committing the value to the GPIO pin using the aforementioned make_change function. In this manner, the valve_control.py IO script handles the actuation of valves and the backing pumps using PV information from the rest of the control system.

Analog Input Signals Script

The gauge controller analog signal processing script on rpi005, named gauge_readoff.py, has the purpose of processing digital data from an MCP3008 ADC circuit that has received analog voltage signals from convectron gauge controllers. Namely, an SPI bus is established in the script and its related pins change to their SPI functionalities. The interpretation of digital data is handled by adafruit libraries in the background through certain SPI objects, and the majority of the python script is dedicated to using the received analog values to perform calculations and set PVs. A modified version of gauge_readoff.py is as follows:

```
from epics import PV
from time import sleep
import busio
import digitalio
import board
import adafruit_mcp3xxx.mcp3008 as MCP
```

```
from adafruit_mcp3xxx.analog_in import AnalogIn
import RPi.GPIO as GPIO

def main():
    ##########################################
    # WRITE INPUT PVs HERE. Nested arrays must be of the form [PV("PV PRESSURE"),
        # PV("PV VOLTAGE"), PV("PV STATUS ON"), CHANNEL #].
    in_pv_list = [[PV("ASP1:CGB:RDVOLT"),PV("ASP1:CGB:RDVAC"),PV("ASP1:CGB:STATON"),0],
                  [PV("ASP1:CG1:RDVOLT"),PV("ASP1:CG1:RDVAC"),PV("ASP1:CG1:STATON"),1],
                  [PV("ASP1:CGA:RDVOLT"),PV("ASP1:CGA:RDVAC"),PV("ASP1:CGA:STATON"),2]]
    # MCP CHIP RESET PV AND PIN CONFIGURATION
    ASP1MCP12RESET = PV("ASP1:MCP1/2:RESET", auto_monitor=True)
    reset_MCP_GPIO_pin = 17
    ########################################

    # initialize a GPIO pin for reset MCP
    GPIO.setmode(GPIO.BCM)
    GPIO.setup(reset_MCP_GPIO_pin, GPIO.OUT)
    GPIO.output(reset_MCP_GPIO_pin, GPIO.HIGH)

    # create spi bus, cs (chip select), mcp objects
    spi = busio.SPI(clock=board.SCK, MISO=board.MISO, MOSI=board.MOSI)
    cs = digitalio.DigitalInOut(board.D5)
    mcp = MCP.MCP3008(spi, cs)

    # initialize mcp channels
    mcp_channels = [MCP.P0,MCP.P1,MCP.P2,MCP.P3,MCP.P4,MCP.P5,MCP.P6,MCP.P7]
    channels = [AnalogIn(mcp,mcp_channels[pv[3]]) for pv in in_pv_list]

    # convert read divided-voltage into pressure
    def rdvoltage_to_pressure(rdvoltage):
        gauge_voltage = 2.0964995*rdvoltage
        gauge_pressure = pow(10,gauge_voltage-4)
        return gauge_pressure

    # MCP chip reset callback function, fixes errors
```

```
43      def reset_MCP(pvname, value, **kwargs):
44          if ASP1MCP12RESET.value == 1:
45              GPIO.output(reset_MCP_GPIO_pin, GPIO.LOW)
46              sleep(0.5)
47              GPIO.output(reset_MCP_GPIO_pin, GPIO.HIGH)
48              ASP1MCP12RESET.put(0)
49      # add reset_MCP function as callback to MCP reset PV
50      ASP1MCP12RESET.add_callback(reset_MCP)
51
52      # main forever-loop, handles pv-pin controls
53      while True:
54          for pv_set in in_pv_list:
55              # get voltage from MCP channel, compute respective pressure, give to PVs
56              rdvoltage = channels[pv_set[3]].voltage
57              pv_set[0].put(rdvoltage)
58              pv_set[1].put(rdvoltage_to_pressure(rdvoltage))
59              # if rdvoltage is ~0, then set the status PV to off
60              if round(rdvoltage,1) == 0:
61                  pv_set[2].put(0)
62              else:
63                  pv_set[2].put(1)
64  main()
```

In the main function, lines 13-18 are used to define necessary arrays and variables for the script. First, the in_pv_list array is declared, which contains sub-arrays whose elements are 3 pyepics PV objects for each gauge: 1 for voltage, 1 vacuum pressure, and 1 for on/off status, and then another element to specify the channel number on the MCP3008 ADC chip that will be accessed for voltage information. Then, an auto-monitored PV object is defined for resetting the MCP chip, where the auto-monitor parameters allows for the support of a later-used callback function. Finally, the variable that stores the GPIO pin number used for resetting the MCP is specified, which is also the pin that delivers power to the MCP. Lines 22-24 set the pin naming convention and initializes the MCP reset pin to output high level voltage, thus powering the MCP. Lines 27-29 create the SPI bus, cs (chip select) and MCP objects, while simultaneously initializing their pins on the RPi board. Lines 32-33 initialize channels on the MCP as being analog-in for every PV-set element in the in_pv_list array. Then, the function rd_voltage_to_pressure is defined, which is called to convert the divided analog voltage that the RPi receives into a vacuum pressure value. It converts by multiplying the read voltage by an approximate division factor from a voltage divider in the circuit, and then uses an exponential function to inverse the logarithmic scale that the gauge controller uses to convert pressure to voltage, thus yielding an accurate vacuum value. The callback function reset_MCP is defined and initialized on lines 42-49. This callback function is used to reset the MCP by turning it off and then on, and is triggered whenever the MCP reset PV changes value. In this callback function, if the MCP reset PV is of value 1, then the MCP reset GPIO pin will turn off (driven LOW) and then on (driven HIGH) – correcting most internal issues of the MCP. Immediately after the function definition, the function is added as a callback to the MCP reset PV, thus calling the reset_MCP function whenever the PV changes value. Followingly, lines 58-68 contain the main run-loop. For each PV-set inside in_pv_list, the corresponding MCP channel's voltage is retrieved and the gauge voltage PV is set to that value, then the vacuum PV value is set to be the result from the rdvoltage_to_pressure function with the read voltage value as the input. Afterwards, the if-else statement behaves so that if the read voltage value is approximately 0, then the gauge must be off, and so the status PV is set to 0 for off; otherwise, the status PV is set to 1 for on. In this manner, the gauge_readoff.py script reads digital voltage values from the MCP ADC that is connected to vacuum gauge controllers, and then determines the corresponding pressure value – setting the respective PVs to these values so the rest of the control system has access to them.

External PV Query Script

The external PV querying script on rpi003, named external_pv_parser.py, has the purpose of retrieving the value of typically inaccessible PVs that are managed on the private ISAC EPICS network, from a TRIUMF HLA web-server. Specifically, the script gets the values of the HEBT1:IMG0:RDVAC and HEBT:IG1:RDVAC PVs from the Jaya web-server https://beta.hla.triumf.ca/jaya. The Jaya web-server uploads PV data as XML files at a subdomain with the name of the PV. Note that few ISAC PVs have been added to this web database, and more must be manually added when needed. The python default package urllib is used to query the web-server and the default package XML is used to parse the XML file for useful data. A modified version of this script is the following:

```python
import urllib.request
from xml.dom import minidom
import time
```

```
4  from epics import *
5
6  ############################################
7  # ADD EXTERNAL PV NAMES HERE.
8  external_pvs = [‘HEBT1:IMG0:RDVAC’, ‘HEBT1:IG1:RDVAC’]
9  ############################################
10
11 # main forever-loop. Handles network requests and updates PVs
12 while True:
13     try:
14         for pv in external_pvs:
15             # open url and open XML file
16             url = "https://beta.hla.triumf.ca/Jaya/get/" + pv
17             xmlf = urllib.request.urlopen(url)
18             dom = minidom.parse(xmlf)
19
20             # get value of external PV
21             pv_value = dom.getElementsByTagName(‘get’)
22             pv_value = pv_value[0].attributes[‘value’].value
23
24             # update local PV with external value
25             if pv_value != ‘Not Monitored Here’:
26                 caput(pv, pv_value)
27                 print(pv, pv_value)
28             time.sleep(10)
29     except Exception as e:
30         print(e)
```

After writing the external PV names in an array, the rest of the script runs in a forever loop. In this loop, all of the web-querying code in placed under a try statement because web requests are unstable and connection errors should not cause the script to end. Then, for each PV in the external_pvs array, lines 16-18 open the corresponding web URL on the Jaya server and then begins parsing the resident XML file, storing it as an object. Then, lines 21-22 retrieves the value in the “value” element under the “get” label, and through the following lines, if that retrieved value is anything but “Not Monitored Here” – the default statement on Jaya for PVs not in the HLA database, then it will set the PV to the retrieved value. Thus, the script external_pv_parser.py sets the value of an ASPIRE EPICS PV to that of the fetched value of an external TRIUMF ISAC PV from the Jaya HLA web-server.

Camera Stream Script

The camera video streaming scripts on rpi003 and rpi004, named stream_camera.py, streams video feed captured from a connected Raspberry Pi Camera to a locally-hosted web-server. This is possible by using the picamera2 library to begin video recording with the camera, and then using web-related libraries to output the video stream to a video file on a locally-hosted website. This website, and the corresponding video stream, is best accessed by entering the address "[IPv4 address]:8000" into a web browser. For instance, the address "10.90.69.233:8000" is currently used to access the video stream from rpi003 over TRIUMF Secure Wi-Fi. In addition to streaming the camera footage, callback functions are employed to continuously adjust the camera settings for exposure and analog gain based on operator-specified PV values. The aspect of the script that initializes the video stream is adapted code written by the Raspberry Pi Organization (https://github.com/raspberrypi/picamera2/blob/main/examples/mjpeg$_s$*erver.py*). A modified version of the stream_camera script on rpi003 is the following:

```
import io
import logging
import socketserver
from http import server
from threading import Condition
from epics import PV
from picamera2 import Picamera2
from picamera2.encoders import JpegEncoder
from picamera2.outputs import FileOutput

##########################################
# STREAM SETTINGS
stream_size = [640,480]
port = 8000
wlanip = "10.90.69.233"
elanip = "169.254.141.56"

# PV declarations
ASP1CAM1STATON = PV('ASP1:CAM1:STATON')
```

```
ASP1CAM1WLANIP = PV('ASP1:CAM1:WLANIP')
ASP1CAM1ELANIP = PV('ASP1:CAM1:ELANIP')
ASP1CAM1EXPOSURE = PV('ASP1:CAM1:EXPOSURE', auto_monitor=True)
ASP1CAM1GAIN = PV('ASP1:CAM1:GAIN', auto_monitor=True)

# HTML code for generated website
PAGE = '''\
<html>
<head>
<title>ASPIRE RPI003 Stream</title>
</head>
<body>
<h1>ASPIRE RPI003 Stream</h1>
<img src="stream.mjpg" width="''' + str(stream_size[0]) +
                  '''" height="''' + str(stream_size[1]) + '''" />
</body>
</html>'''
##########################################

# Video streaming output class
class StreamingOutput(io.BufferedIOBase):
    def __init__(self):
        self.frame = None
        self.condition = Condition()
    def write(self, buf):
        with self.condition:
            self.frame = buf
            self.condition.notify_all()

# Web request and content (video) serving class
class StreamingHandler(server.BaseHTTPRequestHandler):
    def do_GET(self):
        if self.path == '/':
            self.send_response(301)
            self.send_header('Location', '/index.html')
            self.end_headers()
        elif self.path == ‘/index.html’:
            content = PAGE.encode(‘utf-8’)
            self.send_response(200)
            self.send_header(‘Content-Type’, ‘text/html’)
            self.send_header(‘Content-Length’, len(content))
            self.end_headers()
            self.wfile.write(content)
        elif self.path == ‘/stream.mjpg’:
            self.send_response(200)
            self.send_header(‘Age’, 0)
            self.send_header(‘Cache-Control’, ‘no-cache, private’)
            self.send_header(‘Pragma’, ‘no-cache’)
            self.send_header(‘Content-Type’, ‘multipart/x-mixed-replace; boundary=FRAME’)
            self.end_headers()
            try:
                while True:
                    with output.condition:
                        output.condition.wait()
                        frame = output.frame
                    self.wfile.write(b‘--FRAME\r\n’)
                    self.send_header(‘Content-Type’, ‘image/jpeg’)
                    self.send_header(‘Content-Length’, len(frame))
                    self.end_headers()
                    self.wfile.write(frame)
                    self.wfile.write(b‘\r\n’)
            except Exception as e:
                logging.warning(‘Removed streaming client %s: %s’, self.client_address, str(e))
        else:
            self.send_error(404)
            self.end_headers()

class StreamingServer(socketserver.ThreadingMixIn, server.HTTPServer):
    allow_reuse_address = True
    daemon_threads = True

# Exposure and Gain settings callback function. Called when Exposure or Gain PV changes
```

```
 92 def CameraConfig(pvname, value, **kwargs):
 93     picam2.set_controls({"ExposureTime": int(ASP1CAM1EXPOSURE.value),
 94                                              "AnalogueGain": float(ASP1CAM1GAIN.value)})
 95 # setting callbacks
 96 ASP1CAM1EXPOSURE.add_callback(CameraConfig)
 97 ASP1CAM1GAIN.add_callback(CameraConfig)
 98 
 99 # Initializing picam2 video recording
100 picam2 = Picamera2()
101 picam2.configure(picam2.create_video_configuration(main={"size": (stream_size[0],
102                                 stream_size[1])}))
103 output = StreamingOutput()
104 picam2.start_recording(JpegEncoder(), FileOutput(output))
105 
106 # main runloop. Initiatiates web server and updates diagnostic PVs.
107 while True:
108     try:
109         # begin web server and hence video stream, and set PVs
110         address = (‘’, port)
111         server = StreamingServer(address, StreamingHandler)
112         ASP1CAM1STATON.put(1)
113         ASP1CAM1WLANIP.put(wlanip + ":" + str(port))
114         ASP1CAM1ELANIP.put(elanip + ":" + str(port))
115         server.serve_forever()
116     except Exception as e:
117         # update diagnostic PVs in exception event
118         ASP1CAM1STATON.put(0)
119         ASP1CAM1WLANIP.put("NONE")
120         ASP1CAM1ELANIP.put("NONE")
121         print(e)
```

In this script, lines 13-23 define configurable variables for the camera stream, such as the port on which the stream is output from, the IP addresses to display to the operator (note that this is not a functional setting that changes where the stream is hosted), and the variable PAGE which contains the HTML code that structures the stream website. After these configurable variable definitions, lines 39-88 create the website with the video stream output and handle basic web requests. They should never need to be changed. Specifically, lines 39-46 define the StreamingOutput class, which is the data structure later used to direct the video output to the web page. Lines 49-84 define the StreamingHandler class, which handles the web-requests from the user, including automatically directing them to the index page of the web directory and serving the video content. The final class, StreamingServer, is later used to create the web server, and sets StreamingHandler to manage web requests. Followingly, lines 91-93 define the callback function CameraConfig which, when the PV for exposure or analog gain changes, the function is called which re-configures the camera settings with new operator-specified exposure and gain values. The next two lines add the CameraConfig function as a callback to the two auto-monitored PV objects: ASP1CAM1EXPOSURE and ASP1CAM1GAIN. The camera is then initialized and begins recording video, through lines 99-103, with the video output directed to StreamingOutput. Lines 106-120 then contain the main run-loop, which starts the web server and updates PV values accordingly, again changing the PVs to indicate if the server is off in the event an significant error occurs. Note the presence of the "try" statement in the forever loop. This keeps the script running forever, even if errors occur, as web connections are inherently unstable. Thus, the script stream_camera.py on rpi003 streams video camera footage to a web page that users can access. Although the streaming script on rpi003 has been exclusively discussed, the same-named script on rpi004 functions nearly identically.

3.4 Operator Connection to Raspberry Pi OS

An operator will eventually need to gain access to the OS of a physically installed ASPIRE RPi IOC, whether to safely power off the RPi, transfer files to update the IOC software, or change system settings. There are 3 main ways to gain access to an RPi's OS. These ways include: a wired connection to peripheral devices that allow visual access and interaction with the OS, a remote network connection through SSH that allows access to a shell terminal, and lastly a remote network connection through VNC that allows for visual access and interaction with the OS.

3.4.1 Wired Connection

An operator can access an RPi's OS using wired connections. Several cable connections must be made to permit both visual and interactive access to the RPi. An HDMI cable with one end connected to a computer display monitor, and the other end connected to the HDMI port of the RPi, will display the RPi OS once the RPi has been rebooted. Then, a USB mouse and USB keyboard can be connected to the

USB ports of the RPi to allow text and mouse inputs into the OS. With these 3 connections, the RPi OS can be viewed and interacted with. A USB stick can also be plugged into a USB port of the RPi, allowing for file transfers and hence updating of the IOC software.

3.4.2 Remote SSH Connection

An operator can access an RPi's shell terminal remotely over SSH from another computer. For this to be possible, the RPi must be on the same LAN network as the operator's computer, such as the same Wi-Fi network or E-LAN, and the SSH system setting must be enabled on the RPi. Then, on the other computer, the operator can input the terminal command "ssh aspire@[address]", where [address] is an accessible IPv4, IPv6, or MAC address. Once executed, the terminal window converts into a shell terminal of the RPi where commands can be executed. Many packages use SSH to offer additional functionality beyond opening a shell terminal, such as the package scp, which transfers files. Scp is used as "scp [-r] [source file directory] aspire@[address]:[full destination file directory]", where [-r] represents the recursive scp option that transfers folders instead of files. This is useful for updating the IOC software. Thus, SSH offers a remote connection method to an RPi from another computer on the same network.

3.4.3 Remote VNC Connection

For a graphical remote connection to the RPi OS, the operator can connect from another computer using VNC. VNC streams an RPi's desktop while mirroring mouse and keyboard commands from the access computer back to the RPi. To use VNC, the VNC system setting must be enabled on the RPi. Then, the RealVNC Viewer application (https://www.realvnc.com/en/connect/download/viewer/) must be installed on the access computer. Once the app is installed, the target RPi can be accessed by typing its IPv4 address into the top address search bar. Then, a menu will present itself that requests the RPi username and password. Once complete, the VNC connection to the RPi will establish with a desktop windowing appearing, allowing the system to be interacted with remotely. Files can also be transferred using RealVNC viewer, which is useful for updating the IOC software.

4 Conclusion

Through all the instructions contained in sections 2 and 3, the software for the ASPIRE Control System network was programmed and implemented. In section 2, the structure of the software on the operator computer was discussed. Namely, the use of an Ubuntu subsystem with EPICS, EDM, and StripTool installed, and then the use of EDM to build the ASPIRE control system GUI. Followingly, section 3 explained how the ASPIRE RPi IOCs were set up and how the OS was configured, including network settings. Then, the call structure of control system scripts on boot-up was described, along with an explanation of the functioning and logic behind each script. Finally, the recommended way to access the OS on each RPi, including remote methods, were outlined. In this way, the ASPIRE control system was built, and can be replicated or expanded upon by following similar steps.

References

Henders, M et al. (June 18, 2024). *Raspberry Pi Cluster for EPICS Controlled Tabletop Simulations.*

Sinclair, John (2007). *Extensible Display Manager.* URL: `https://controlssoftware.sns.ornl.gov/edm/eum.html`.