

Programming ACS-Based PI Motion Controllers with Python

Adding more Functionality to PI's Ethercat-based ACS Controllers

This sample program is intended to be an approachable demonstration of basic Python programming for communicating with PI's ACS-based motion controllers. Python is a flexible, simple, and accessible programming option that has a strong community. Combining the power of PI hardware with the availability and strengths of Python further enhances your ability to deploy world-leading precision automation. This post will walk you through building a basic Graphical User Interface (GUI) for communication with an ACS-based PI controller or simulator.

Useful Documentation and Links

PI

- [G-901 Motion Controller for High Power Requirement](#)
- [G-910.RC02 Motion Controller for Low Power Requirement](#)
- [G-910.RC01 • RC03 ACS Driver Module for Low Power Requirement](#)
- [A-81x Plglide Motion Controller for 1, 2 or 4 Axes](#)
- [A-82x Plglide Motion Controller for 4, 6 or 8 Axes](#)

Python

- <https://www.python.org/downloads/>
- <https://code.visualstudio.com/docs/python/python-tutorial>
- <https://tkdocs.com/tutorial/index.html>
- <https://docs.python.org/3/library/tkinter.html>
- <https://tcl.tk/man/tcl8.6/TkCmd/contents.htm>



ACS

- [SPiiPlus ADK Suite](#) (Registration Required)
- Typical install location: C:\Program Files (x86)\ACS Motion Control\
 - Python Library: \SPiiPlus ADK Suite v3.14\SPiiPlus Python Library
 - Python Guide: \SPiiPlus Documentation Kit\Software Guides\SPiiPlus-Python-Library-Reference-Programmers-Guide.pdf
 - Simulator and User Mode Driver: \SPiiPlus Documentation Kit\Software Guides\SPiiPlus-Utilities-User-Guide.pdf

Basic Python Setup


Skip this section if you already have the Python interpreter and an editor. This guide uses VS Code and will require tk and SPiiPlusPython libraries.

1. Download and install Python (SPiiPlusPython supports 3.10, 3.11, & 3.12)
 - <https://www.python.org/downloads/>

2. Download and install ACS's SPiiPlus ADK Suite
 - <https://www.acsmotioncontrol.com/resources/downloads/?product-id=spiiplus-mmi-application-studio>
3. Download, install, and configure Microsoft Visual Studio Code (VS Code) for Python
 1. <https://code.visualstudio.com/download>
 2. Run the installer with default settings, then launch VS Code
 3. Go to the Extensions view by clicking on the square icon on the sidebar or pressing **Ctrl+Shift+X**.
 4. Search for "Python" in the extensions search bar.
 5. Install the extension authored by Microsoft (usually the first result).
 6. For more info, see documentation section at the end of the post.
4. Install SPiiPlusPython library (requires pip and wheel dependencies)
 1. Navigate to the install directory for ACS and find the SPiiPlusPython wheel (.whl) file version that matches your Python version (cp3xx)
 - Typically C:\Program Files (x86)\ACS Motion Control\SPiiPlus ADK Suite v3.14.01\SPiiPlus Python Library (or similar)
 2. Open the terminal in VS Code (Ctrl+`)
 3. Install the package with `pip install "path"`
 - eg `pip install "C:\Program Files (x86)\ACS Motion Control\SPiiPlus ADK Suite v3.14.01\SPiiPlus Python Library\SPiiPlusPython-3.14.0.0-cp312-cp312-win_amd64.whl"`
 4. Close terminal

Step-by-step Guide for connecting to a controller via Python GUI

Prerequisites:

- Python and editor installed (VS Code used in this guide)
- Tkinter is installed in active environment
- Check that SPiiPlus User Mode Driver (UMD) is running. The SpiiPlus Simulator will launch automatically with any attempt to connect to it, assuming the UMD is running.
 - Look at sections 2 and 4 of the SPiiPlus-Utilities-User-Guide.pdf for info on the SPiiPlus User Mode Driver and SPiiPlus Simulator.
 - Check if the UMD is running by looking at the system tray for the icon () or by looking for the background process 'ACS Motion Control User-Mode-Driver' in the Task Manager. If it is not running, then start it via the Start Menu (search ACS, or user-mode), or by navigating to your ACS installation (typically, C:\Program Files (x86)\ACS Motion Control\SPiiPlus Runtime Kit\User Mode Driver) and running the ACSCSRV.exe application.
 - Open the UMD window by right-clicking the icon in the system tray and selecting 'Open'.
 - The SPiiPlus simulator emulates programming features of the controller, with some exceptions.
 - Connecting to the simulator can be done via MMI Application Studio, or via COM, .NET, or Python libraries.
 - The simulator connection has a specific connection command (OpenCommSimulator), but can also be connected via the standard TCP/IP command (OpenCommEthernetTCP). When using MMI Application Studio, or a host application, connecting to a local PC IP will connect to the simulator. For the sake of simplicity, this guide will use the TCP command, so that the program is agnostic to the use of the simulator or a physical controller.

- Get local IP address if using the simulator (Any local IPv4 IP will work, if you have multiple adapters or VPN), or use the default IP of 10.0.0.100 if using a physical controller.
 - Grab IP from UMD (simulator tab, under 'Network hardware found')
 - Or from command line via `ipconfig`
 - Or from 'View Your Network Properties' in Start Menu
- SPiiPlusPython library is installed in active environment

Procedure:

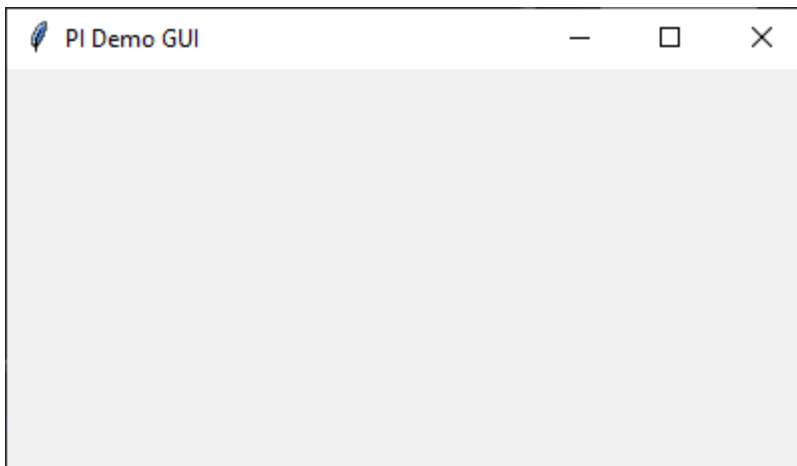
1. We'll start by creating a simple program for a GUI using tkinter.

1. Create a new .py file named main.py. Open the file in VS Code and enter the following. Save the file.

```
import tkinter as tk
app = tk.Tk()
app.title("PI Demo GUI")
app.geometry("400x200")
app.mainloop()
```

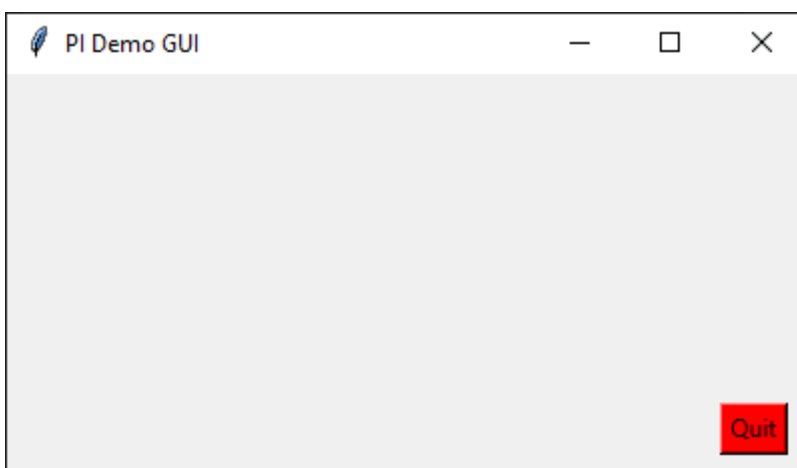
2. Run the program, observe the application that opens. Let's connect the code with the window.

- The first line imports the tkinter module (library) as an alias called 'tk'. Importing a module allows us to use functions, classes, variables, or other code contained in that module. The alias 'tk' makes the code more readable and simpler to write (without the alias, line two would be: `app = tkinter.Tk()`). The Tkinter module is for fast, simple GUI generation.
- The second line creates an instance of the class 'Tk' (from the Tkinter library) called 'app' which is an object that serves as the main window of the application on which widgets and other GUI elements are placed.
- The third and fourth lines add a title and size. In this code, `.title` and `.geometry` are methods of the Tk class (remember, 'app' is an instance of the Tk class). More of these methods and other training can be found in the Tk documentation or in other online tutorials.
- The fifth line calls the mainloop method on the main window (defined as 'app'); 'mainloop()' opens the window and keeps it open and able to respond to user interactions or events and inputs until the application is closed. In the context of this guide, `app.mainloop()` is always the last line in the main.py file.



3. Close the window (before running, always ensure the app is closed from the last execution). We'll now add a button (a button is a tk *widget*; others include frame, label, and entry, among many more), with the code below. Run and observe.

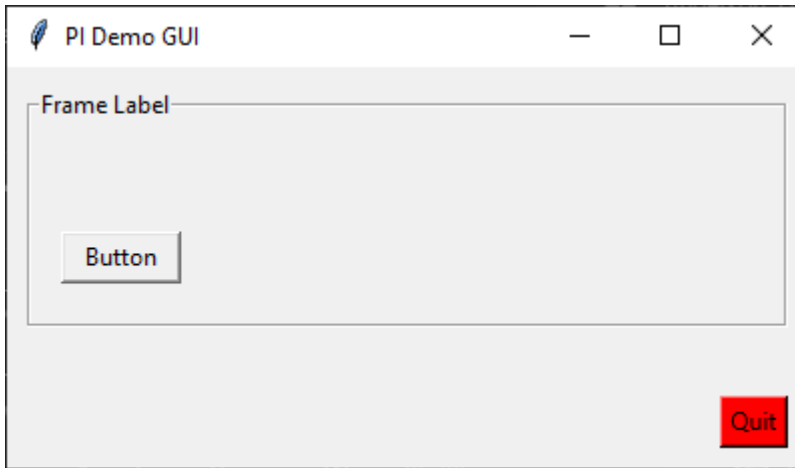
```
buttonQuit = tk.Button(app, text="Quit", command=app.quit, bg="red",
fg="black")
buttonQuit.place(relx=1.0, rely=1.0, x=-10, y=-10, anchor="se")
```



2. Now that we have a window, we'll add a frame. A frame is a widget that acts as a container to organize and group other widgets (eg, buttons, labels, text entries, other frames, etc).

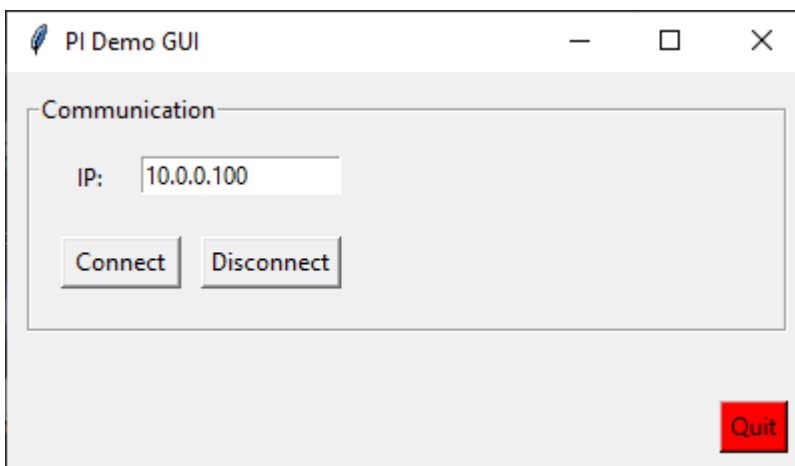
1. Add the lines below to main.py, under the quit button. Other widgets can be positioned relative to the frame, in this case a button. The last two lines, below the frame, create another button. This button is tied to 'frameBasic' instead of 'app'. The parent widget (app, frameBasic) defines the position of the child widget (buttonBasic).

```
frameBasic = tk.LabelFrame(app, text="Frame Label", width=380,
height=120, padx=5, pady=5)
frameBasic.place(x=10, y=10)
buttonBasic = tk.Button(frameBasic, text="Button")
buttonBasic.place(x=10, y=50, width=60)
```



2. Change the button name to 'buttonConnect' and add a second button placed 70 pixels to the right (positive) and 70 pixel wide that is called 'buttonDisconnect'. Also add an entry with a label (code shown below). There is no back end for this yet, so the buttons don't do anything when clicked.

```
entryIPText = tk.StringVar(value="10.0.0.100")
entryIP = tk.Entry(frameBasic, textvariable=entryIPText)
entryIP.place(x=50,y=10, width=100)
entryIPLabel = tk.Label(frameBasic, text="IP:")
entryIPLabel.place(x=15,y=10)
```



3. Add a backend to Open and Close a Connection

1. In main.py, import the SPiiPlusPython module as alias sp with the command: `import SPiiPlusPython as sp`. Then change the tk button to add a 'command' to open the connection and run the program. If you see the error `'ModuleNotFoundError: No module named 'SPiiPlusPython'`, then double check your active environment (ensure the right env is active and that SPiiPlusPython is installed). In VS Code, you can select the active environment with `Ctrl+Shift+P` and then `Python: Select Interpreter`, for more info see <https://code.visualstudio.com/docs/python/python-tutorial>.

```
# Simulator (shown for example only, not used in this guide)
buttonConnect = tk.Button(frameBasic, text="Connect",
```

```

command=sp.OpenCommSimulator()
# Physical controller (10.0.0.100 is the default ACS IP), or connect to
simulator with your local IP
buttonConnect = tk.Button(frameBasic, text="Connect",
command=sp.OpenCommEthernetTCP("10.0.0.100", 701))

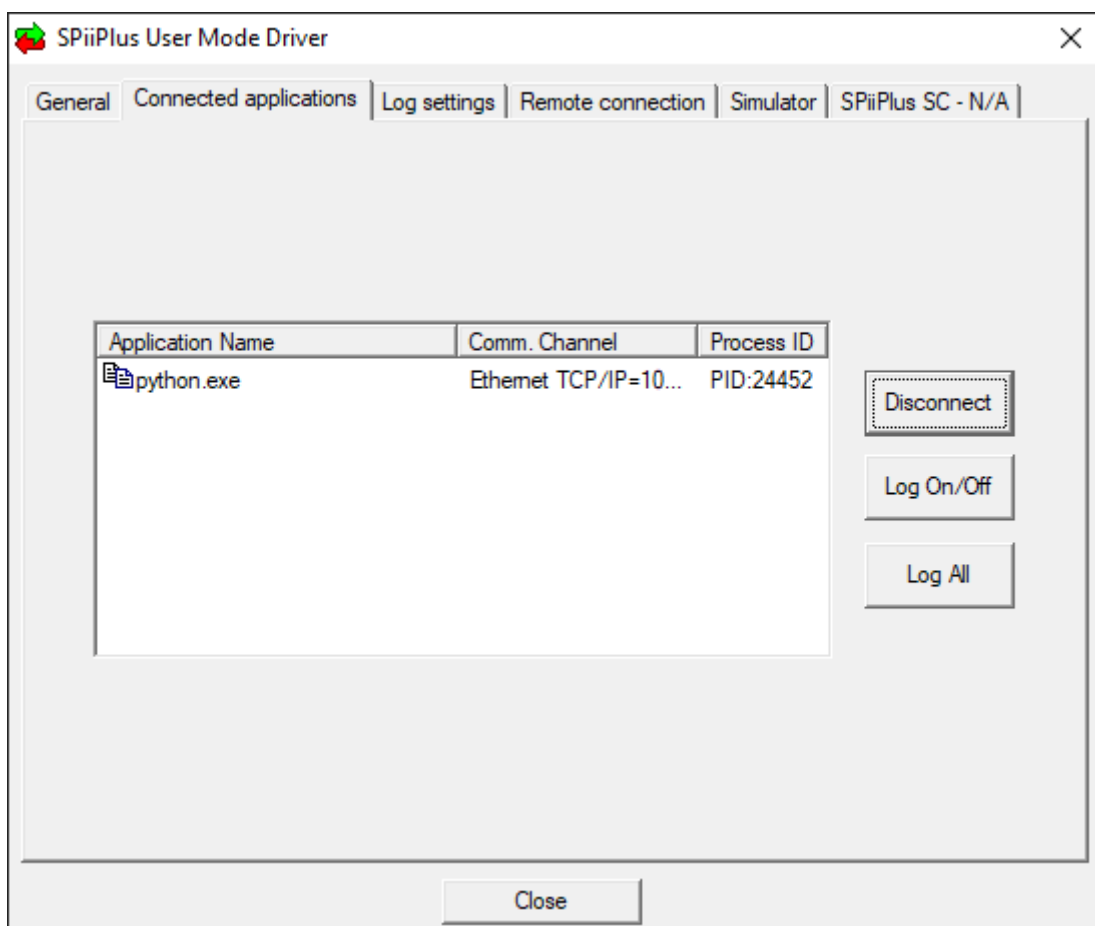
```

2. Now we can connect to the controller, but notice that the controller connects on program execution (you can see this in the UMD under connected applications, as below). To solve this, we can use 'lambda' to wrap the function call so that it is called on button press, not during the creation of the button, as it provides a way to pass arguments to functions when used with Tkinter 'command' callbacks. Add `lambda:` as the command as below (remember you need your own local IP if you're using the simulator) and observe the behavior when executed. The connection is now opened on button press, not program execution.

```

buttonConnect = tk.Button(frameBasic, text="Connect", command=lambda:
sp.OpenCommEthernetTCP("10.0.0.100", 701))

```



3. Now that we can connect on button press, we'll add a disconnect command to the disconnect button, so that we can disconnect without closing the application. `CloseComm` requires a communication handle to close the connection so 0 is used here, but we'll add a comm handle variable in the next steps. Don't forget lambda so that we can correctly pass arguments to the `CloseComm` function.

```
buttonDisconnect = tk.Button(frameBasic, text="Disconnect",
command=lambda: sp.CloseComm(0, failure_check=True))
```

4. Communication Handle - Functions

1. To close the connection, we need to assign a handle to the open connection. This added complexity pushes us to add a separate function to handle the connection. Let's define a connect function and use the value from the entryIP widget shown above to get the IP from the user (default of 10.0.0.100 is automatically populated). We'll add some global variables at the top of main.py and also declare them in the function (for modification within the local scope of the connect function). Enter the code shown below into main.py in the appropriate locations, including updating the buttons. Run the program and observe that we can connect and disconnect using the buttons. Note that there is a limitation here where channel is updated to the most recent connection, so the disconnect will only work on one connection, though pressing Connect will keep opening connections; all connections are closed on application quit. A more robust solution is outside the scope of this introductory guide, but one such path might be using threading locking to lock out the connect section and then disable the button (so that only one connection is ever open in one GUI).

```
# Global variable definitions in main.py, below import code
port = 701
ip = None
channel = -1
# Connect Function in main.py
def connect():
    global channel, ip, port
    port = 701
    ip = entryIP.get()
    channel = sp.OpenCommEthernetTCP(ip, port)
    if channel != -1:
        print(f"Successfully connected to IP {ip} on channel
{channel}")
    else:
        print(f"Failed to connect to IP: {ip}")
# Updated buttons
buttonConnect = tk.Button(frameBasic, text="Connect", command=lambda:
connect())
buttonDisconnect = tk.Button(frameBasic, text="Disconnect",
command=lambda: sp.CloseComm(channel, failure_check=True))
```

2. Following this example, add a disconnect function. Note that we don't need the global declaration of channel in the disconnect function, as we are not modifying the variable, just reading it. Don't forget to change the disconnect button command to use the new function. We've also added some additional verbose printouts, let's look at the status of these in the terminal under various conditions (listed below, in the order that they occurred). Change the printouts to suit your application.

```
def disconnect():
    try:
        sp.CloseComm(channel, failure_check=True)
        print(f"Disconnected on channel {channel}")
    except Exception as e:
        print(f"Error occurred when trying to disconnect channel
{channel}")
        print(e)
```

- Attempted to connect to 10.0.0.100 when on simulator - the IP is not on the network so the connection fails
- Attempted to disconnect when not connected; no channel -1 (set as the default for the variable 'channel'), so the disconnect fails and I print the ACS error
- Attempted to connect to my local IP, 10.31.16.21, and it is successful
- Attempted to disconnect from last connected channel and it is successful

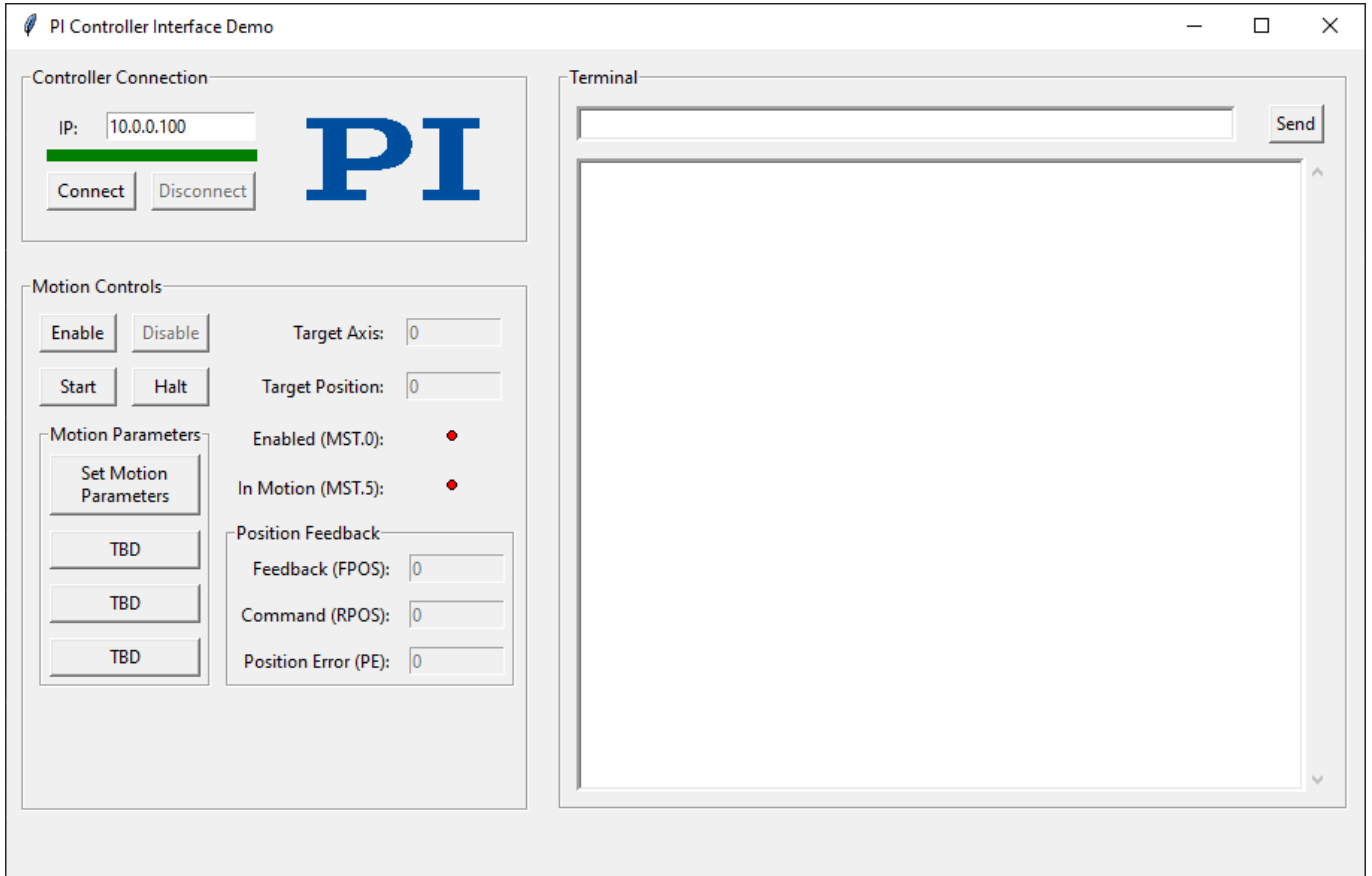
```
Failed to connect to IP: 10.0.0.100
Error occurred when trying to disconnect channel -1
ACS Command error:
Received error code: 197
Message Error: Communication server can't be found
Successfully connected to IP 10.31.16.21 on channel 0
Disconnected on channel 0
```

5. You should now have a functional GUI in main.py, with connect and disconnect buttons and an IP entry field. A summary of the code is below. The next steps are to increase the value of the program's architecture by adding additional frames, functions, and .py files.

1. Import tkinter and SPiiPlusPython as alias tk and sp
2. Global variables port, ip, channel
3. Main tk instance ('app')
4. Quit button
5. Frame for communication
 - IP entry and label
 - Buttons for connect and disconnect
6. Functions for connect and disconnect
7. Mainloop

Final Thoughts

With the addition of a motion frame and a terminal frame, along with graphics, our basic GUI can be expanded like the image below. Please fill out the form to download the sample GUI shown.



PI's A-810 - A-814 multiaxis motion controllers are based on ACS modules