Pyramid Technical Consultants, Inc

1050 Waltham Street, Suite 200
Lexington, MA 02421, USA
Phone: 781.402.1700 Website: www.ptcusa.com

Pyramid Product Documentation

T1 Programmer's Guide

Document ID: 2025914461

Version: 1.1.2

Author: Matthew Nichols

Last modified on: 06/22/2021

Exported by: Matthew Nichols

Exported on: 06/22/2021

Table of Contents

1 Introduction

Document ID: 2025914461

Version	1.1.2
Author	Matthew Nichols

1.1 Purpose

This guide should serve as a starting point for programmers to get started, collecting data and configuring devices. The biggest benefit of using Pyramid devices is that you automatically get the Pyramid software team on your side. If you have questions, bugs, or feature requests, we want to help and we love talking to our customers about their projects. Get in touch with us by opening a ticket through the support portal[1] or just sending us an email at support@ptcusa.com[2].

1.2 Intended Audience

Programmers who are interested in writing code that works with the T1 and corresponding hall probe devices. Software management team members looking to evaluate software integration requirements.

1 https://pyramidtc.atlassian.net/servicedesk/customer/portal/1
2 mailto:support@ptcusa.com

2 Programming Quick Start

2.1 Use CURL To Get Measured Field Value

```
curl -X GET http://<IP ADDRESS>/io/t1/probe/field/value.json
```

2.2 Use Excel To Get Measured Field Value

Enter the following function into a cell, click outside the cell, and use Ctrl+Alt+F9 to refresh the value.

```
=WEBSERVICE("http://<IP ADDRESS>/io/t1/probe/field/value.json")
```

2.3 Use Python, HTTP, And requests[3] To Get Measured Field Value

```
import requests
print("Field =", requests.get("http://<IP ADDRESS>/io/t1/probe/field/
value.json").json(), "Gauss")
```

2.4 Use Python, EPICS, And pyepics[4] To Get Measured Field Value

```
import epics
pv = epics.PV("/t1/probe/field/value")
print("Field =", pv.get(), "Gauss")
```

3 https://docs.python-requests.org/en/master/
4 https://github.com/pyepics/pyepics

3 How Device Data Is Structured

All data and configurations are stored in data structures called an **IO**. All IOs are primitive values (number, string, or boolean) or arrays of primitives. Each IO has a handful of **Fields** associated with them to describe their values and metadata. For example, there could be an IO with the `name` field “voltage”, the `value` field 1.23, the `label` field “Voltage”, and a `units` field “V”.

IO and fields exist on an organized tree structure similar to a file system. Also like a file system, they are referenced by their unique **Path** in the structure. For example `/device/sub_module/voltage/value` would be the path to the value field of an IO with the name field “voltage” whose parent has the name field “sub_module” and whose grandparent name field is “device”.

4 Available Protocols

All fields are stored in a virtual file system on the device. We can read and write to those files in several ways that will be described below.

4.1 HTTP

Devices have an HTTP server built-in that can serve any file on the filesystem include the special field files. Simply using GET and PUT requests you can read and write field files. This method is great if you are already using HTTP or need to implement something quickly to get up and running. Standard tools like curl[5] are invaluable for debugging or even just implementing small scripts.

URLs should be structured like the following `http://<ip address>/io/device/sub_module/voltage/value.json`.

Python GET and PUT example:

```
import requests
print(requests.get("http://<ip address>/io/device/sub_module/voltage/
value.json").json())
requests.put("http://<ip address>/io/device/sub_module/command/value.json", str(command)
)
```

4.2 HTTP Via CURL

cURL[6] is a great choice for debugging or writing a quick and dirty script. Simply enter the following command to GET and PUT a field value.

```
curl -X GET http://<ip address>/io/device/sub_module/voltage/value.json
curl -X PUT -d "1.234" http://<ip address>/io/device/sub_module/command/value.json
```

5 https://curl.se/
6 https://curl.se/

4.3 SFTP

Devices have an SFTP server that clients can use to mount the device drive local to their development machine. This method is particularly useful if you are using a Linux-based operating system or are used to mounting network drives. This method also allows you to use text editors to easily open and view the files before writing your code.

4.4 EPICS

EPICS[7] comes for free when using IGX devices. No need to write your own drivers, the device is an EPICS server all on its own. The PV names for all the fields are just the path for that field. Optionally if you are running multiple of the same device, you can prepend the IP address of the device before the channel name, for example, `192.168.0.5:/device/sub_module/voltage/value` and `192.168.0.6:/device/sub_module/voltage/value`. EPICS is great if you are already using EPICS in your control system. If not, then you may want to look into the other communication methods first, as they will be much easier to get up and running without significant library support.

Python example:

```python
import epics
pv = epics.PV("/device/sub_module/voltage/value")
print(pv.get())
```

4.5 WebSockets

The WebSockets API is what our built-in web GUI uses, and enables streaming data at high rates. Unfortunately, the protocol is still under active development and may be subject to radical changes going forward. If you are still interested please, contact us at support@ptcusa.com[8] and tell us about your project, we want as much user input as possible when designing our protocols.

The protocol, as it stands today, simply exchanges plain JSON structures. One message from the client to the device to establish “subscriptions” to various IO, then another to request the latest data.

7 https://epics.anl.gov/
8 mailto:support@ptcusa.com

5 IO Tables

5.1 Probe Data And Configuration

Path	Units	Type	Direction	Notes
`/t1/probe/field`	Gauss	Number	Readonly	The measured magnetic field at the probe tip.
`/t1/probe/average_field`	Gauss	Number	Readonly	The measured field with extra averaging. Useful for display purposes.
`/t1/probe/average_temperature`	Celsius	Number	Readonly	The average temperature at the probe tip. Useful for doing your own temperature monitoring.
`/t1/probe/offset`	Gauss	Number	Read/Write	User settable field offset. Useful for zeroing before measurements.
`/t1/probe/connected`	-	Boolean	Readonly	True if the probe is properly connected to the device. Use it for sanity checking.

5.2 T1 Configuration

Path	Units	Type	Direction	Notes
`/t1/configuration/range`	-	String	Read/Write	Set’s the programmable gain for the measurements. Possible values are “1x”, “4x”, “10x”, and “40x”.

Path	Units	Type	Direction	Notes
`/t1/configuration/rate`	Hertz	String	Read/Write	The data collection and averaging rate. Possible values are “10”, “50”, “100”, “500”, “1000”, “5000”, and “25000”.

6 Practical Code Examples

6.1 Read Field Value Using Python And HTTP

A super simple example to show how you can collect the measured field value. For simplicity, all the methods are called inline. In production code, you should create wrapper functions to reduce your code complexity.

```
import requests

# Device IP address
ip = "192.168.55.239"

# The target URL to make our request
url = "http://" + ip + "/io/t1/probe/field/value.json"

# Send our GET request and parse the resulting JSON value
print("Field =", requests.get(url).json(), "Gauss")
```

6.2 Get Field Value Using Python And EPICS

This example uses the Python package pyepics[9]. Note that the IP address is not required if you are using a single T1. If you have multiple on the network you will need to prepend the IP address in the channel name. For example `192.168.0.5:/t1/probe/field/value`.

```
import epics

# Create a PV object for the field
pv = epics.PV("/t1/probe/field/value")

# Get the current field value
print("Field =", pv.get(), "Gauss")
```

9 https://github.com/pyepics/pyepics

6.3 Programatically Zeroing The Probe Using Python And HTTP

A more complicated example that shows how to get data, and set configurable values. Programatically zeroing a probe is a common procedure before doing a relative measurement.

```
import requests
import time

# Device IP address
ip = "192.168.55.239"


# Helper function that returns the current field measurement
def GetField():
    return requests.get("http://" + ip + "/io/t1/probe/average_field/value.json").json()


# Helper function that sets the device offset to the given value
def SetOffset(offset):
    return requests.put("http://" + ip + "/io/t1/probe/offset/value.json", str(offset)).
json()


print("Zeroing field probe")

# First we get rid of any existing offset, by setting it to zero and waiting
SetOffset(0.0)

# Wait for the new offset to propagate to the new data
time.sleep(0.5)

# Get the current field.
# Set the offset to the previously measured field, effectively zeroing it.
SetOffset(GetField())

# Wait for the new offset to propagate to the new data
time.sleep(0.5)

# Get the field again to confirm the zeroing worked.
print("Newly zeroed field", GetField(), "G")
```

6.4 Programatically Zeroing The Probe Using Python And EPICS

Same example as above but using EPICS.

```
import epics
import time

# Create our PV objects
field = epics.PV("/t1/probe/average_field/value")
offset = epics.PV("/t1/probe/offset/value")

print("Zeroing field probe")

# First we get rid of any existing offset, by setting it to zero and waiting
offset.put(0.0)

# Wait for the new offset to propagate to the new data
time.sleep(0.5)

# Get the current field.
# Set the offset to the previously measured field, effectively zeroing it.
offset.put(field.get())

# Wait for the new offset to propagate to the new data
time.sleep(0.5)

# Get the field again to confirm the zeroing worked.
print("Newly zeroed field", field.get(), "G")
```

6.5 Collect Full Data Rate Field Data And Write To CSV Using Python And WebSockets

This example is considerably more involved but allows you to stream full-speed device data and collect it to a CSV file.

```
import websocket
import time
import json
import csv

ip = "192.168.55.239"  # Device IP address
collection_time = 2.0  # Seconds to collect data
output_file = "t1_data.csv"  # Data output file

# Database for storing collected data
database = {
    "/t1/probe/field/value": []
}

# Create the WebSocket, uses port 80 by default
ws = websocket.create_connection("ws://" + ip)


# Sends the device an event structure
# Optionally contains a payload called data
def sendEventData(event, data=None):
    # Convert dictionary to JSON and send
    ws.send(json.dumps({"event": event, "data": data}))


# Subscribe to the IO fields we are intereseted in
# In this case it is just the field value but there could be more
# The boolean value indicates wether the data should be buffered or not
# Buffered data means that all samples are send to the client on a get event
# Unbuffered data means that only the most recent sample is sent on a get event
def sendSubscribeEvent():
    sendEventData("subscribe", {
        "/t1/probe/field/value": True
    })


# Request the device sends us the new data it has collected
# since the last time we sent a get event.
def sendGetEvent():
    # No data needed for the get event if you have already
    # previously sent the subscribe event message
    sendEventData("get")
```

```
# Response event handler, called every time we get a response
# from the device. Handles the processing of newly collected data
def onMessageEvent(event, data):
    # Check to make sure the response is an update event
    # Update events carry our subscription data
    if (event == "update"):
        # The dictionary contains all the values for each path
        for (path, values) in data.items():
            # Append the new values to the local database
            database[path] += values
        # Send another get event to request more data
        sendGetEvent()


print("Starting collection at", ip, "for", collection_time, "seconds")

# Send an initial subscription event and get event
# in order to start the collection process
sendSubscribeEvent()
sendGetEvent()

# Remember the start time
start = time.time()

# Collect data for a given time
while time.time() - start < collection_time:
    # Wait for a responses from the device
    response = json.loads(ws.recv())
    # Process the received event and data
    onMessageEvent(response["event"], response["data"])

# Once we've finished collecting data we can process
# it however we like. In this case we write it to a CSV file
with open(output_file, "w", newline="") as file:
    writer = csv.writer(file, delimiter=",", quotechar="\"",
                        quoting=csv.QUOTE_MINIMAL)
    writer.writerow(["Values", "Timestamps"])

    value_pairs = database["/t1/probe/field/value"]

    for (value, time) in value_pairs:
```

```
        writer.writerow([value, time])

    print("Collected", len(value_pairs), "samples, written to", output_file)


# Close our connection
ws.close()
```

7 Best Practices

- **Keep IO paths parameterized** - Paths might change in future versions of firmware as the API evolves. Parameterize your path variables to keep your code flexible.
- **Reuse connections when possible** - Reuse your sockets if you want to make multiple requests, the firmware supports recycling TCP connections for HTTP and WebSockets. The resulting code will be more performant.
- **Make your own wrapper functions** - No generic API can ever beat the convenience of custom wrappers specifically made for your application. All the IO behaves the same way, and that lends itself well to being generalized.
- **When possible, decouple from a specific protocol** - HTTP might suit your needs well today, but maybe down the line you want to use EPICS instead. Write your code in such a way that it makes it possible to switch between either.
- **Ask for help** - Pyramid is here to help you. Your feedback drives how we develop our interfaces in the future. Send any questions you have to support@ptcusa.com[10]

10 mailto:support@ptcusa.com