T1 (18014)

T1 - Programmer Manual

Table of Contents

1 Introduction

Document ID: 2025914461

Version	1.1.4
Author	Matthew Nichols

2 Introduction

Welcome to the T1 programmer manual - your primer for leveraging Pyramid devices for data collection and device manipulation. This guide aims to provide a comprehensive foundation for programmers keen on exploring the nuances of the T1 and associated hall probe devices.

One key benefit of using Pyramid devices is the immediate access to our dedicated software support team. Whether you hit a roadblock, need clarification, or have an innovative feature request, our team is available and enthused to assist. We thrive on engaging with our clients about their projects, providing support at every stage of the process.

This guide is a valuable resource not only for programmers but also for software management professionals assessing software integration requirements. We've laid out essential information to kickstart your work. However, should you need further assistance or have additional queries, our support portal is always available. Alternatively, you can reach us via email at support@ptcusa.com.

We appreciate your decision to adopt Pyramid devices and eagerly anticipate collaborating with you on your programming journey.

This manual specifically delves into the programming exclusive to the T1 product. For a broader understanding of the IGX software framework, including details on all network protocol options, we recommend checking out both the user and programmer manuals available on the IGX product page.

You can access these resources at: https://ptcusa.com/products/igx .

3 Programming Quick Start

To retrieve a measured field value using cURL, use the following command:

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

To retrieve a measured field value using Excel, enter the following function into a cell and press Ctrl+Alt+F9 to refresh the value:

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

To retrieve a measured field value using Python and the `requests` library, use the following code:

```python
import requests

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

To retrieve a measured field value using Python and the `pyepics` library, use the following code:

```python
import epics

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

4 Device Data Structure

All data and configurations on Pyramid devices are stored in data structures called IOs. An IO can be a primitive value (such as a number, string, or Boolean) or an array of primitives. Each IO has several associated fields that describe its value and metadata. For example, an IO could have a name field "voltage", a value field of 1.23, a label field of "Voltage", and a units field of "V".

IOs and fields are organized into a tree structure similar to a file system, and they are referenced by their unique path in the structure. For example, the path `/device/submodule/voltage/value` would refer to the value field of an IO with the name field "voltage" whose parent has the name field "submodule" and whose grandparent has the name field "device".

5 Available Protocols

Pyramid devices utilize a virtual file system to store all fields, and we can read and write to these files using various protocols. In this section, we will describe the available protocols and how to use them.

5.1 HTTP

Pyramid devices have a built-in HTTP server that can serve any file on the virtual file system, including field files. Using GET and PUT requests, we can read from and write to field files. This method is useful if you are already using HTTP or need to implement something quickly to get started. Standard tools like cURL can be used for debugging or even implementing small scripts.

To access a field file using HTTP, use the following URL structure:

`http://<ip address>/io/device/sub_module/voltage/value.json`.

Here's an example of using Python's requests library to GET and PUT a field value:

```python
import requests

# GET field value
response = requests.get("http://<ip address>/io/device/sub_module/voltage/value.json")
value = response.json()
print(value)

# PUT field value
command = 1.234
requests.put("http://<ip address>/io/device/sub_module/command/value.json", str(comma
nd))
```

5.2 HTTP via cURL

cURL is a powerful tool for debugging and writing quick scripts to access field values. To GET or PUT a field value using cURL, use the following commands:

```
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
```

These commands will GET and PUT the field values, respectively.

5.3 SFTP

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

To use SFTP, you'll need to connect to the device using an SFTP client. Once connected, you'll have access to the entire virtual file system on the device. You can edit the files directly on your local machine, and the changes will be reflected on the device.

This method can be particularly useful for debugging or making quick edits to the field files. However, it's important to keep in mind that this method does require a bit more setup than HTTP or cURL, and it may not be suitable for all use cases.

5.4 EPICS

When using IGX devices, EPICS comes for free and there is no need to write your own drivers. The device is an EPICS server on its own, and the PV names for all the fields are simply the path for that field. Optionally, if you are running multiple devices of the same type, 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 a great option if you are already using EPICS in your control system. However, if you're not already using EPICS, it may be easier to look into other communication methods first, as they will likely be much easier to get up and running without significant library support.

In summary, EPICS provides a convenient way to access field values on IGX devices without needing to write your own drivers. However, it's important to weigh the benefits and drawbacks of using EPICS compared to other communication methods before deciding which method to use.

Python example:

```python
import epics

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

5.5 WebSockets

The WebSockets API is used by our built-in web GUI and enables streaming data at high rates.

If you're interested in using WebSockets, please contact us at support@ptcusa.com and tell us about your project. We value user input when designing our protocols.

As it stands today, the WebSockets API simply exchanges plain JSON structures. One message is sent from the client to the device to establish "subscriptions" to various IO, followed by another message to request the latest data.

There is a complete example of using WebSockets at the end of this manual.

6 Available IO

6.1 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.

6.2 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”.
`/t1/configuration/rate`	Hertz	String	Read/ Write	The data collection and averaging rate. Possible values are “10”, “50”, “100”, “500”, “1000”, “5000”, and “25000”.

7 Practical Code Examples

7.1 Read field value using Python and HTTP

Here's a super simple example that demonstrates how to collect a measured field value using the HTTP protocol and the `requests` Python library:

```python
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
response = requests.get(url)
value = response.json()
print("Field =", value, "Gauss")
```

This example sends a GET request to retrieve the value of the `field` IO associated with the `t1/probe` device. The resulting JSON value is then parsed and printed to the console.

Note that in production code, it's generally a good idea to create wrapper functions to reduce code complexity and improve readability.

7.2 Get field value using Python and EPICS

Here's an example that uses the `pyepics` Python package to collect a measured field value:

```python
import epics

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

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

This example creates a PV object for the `field` IO associated with the `t1/probe` device. The current value of the field is then retrieved using the `pv.get()` method, and printed to the console.

Note that if you have multiple T1 devices on the network, you will need to prepend the IP address in the channel name, like so: `192.168.0.5:/t1/probe/field/value`.

7.3 Programmatically zeroing the probe using Python and HTTP

Here are examples of how to programmatically zero the probe using Python and HTTP or Python and EPICS:

Python and HTTP Example:

```
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").jso
n()

# 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")
```

Same example as above but using EPICS.

```
import epics
import time
```

```python
# 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")
```

These examples demonstrate how to zero the probe by setting the device offset to the current field measurement. This is a common procedure before doing a relative measurement. The examples use the `requests` library for HTTP communication and the `epics` library for EPICS communication. Note that in production code, it's generally a good idea to create wrapper functions to reduce code complexity and improve readability.

7.4 Programmatically Setting the Range

In this example we set the T1 measurement range to one of the 4 possible values. The ranges are defined as gains, where the more gain you include the smaller the maximum field you are able to measure.

Possible values are “1x”, “4x”, “10x”, and “40x”.

```
import requests
import json

# Device IP address
ip = "192.168.55.239"

def SetRange(range):
    return requests.put("http://" + ip + "/t1/configuration/range", json.dumps(range)
).json()

# Set the range
SetRange("10x")
```

The `SetRange` function is a helper that sets the T1 device range by sending an HTTP PUT request to the appropriate endpoint on the device. We use the `json.dumps()` function to convert the range value to a JSON formatted string for transmission. The range is passed as an argument when the function is called. In this case, we set the range to "10x".

7.5 Programmatically Setting the Data Rate

In this example we set the T1 data rate to one of the 7 possible values. The data rate is defined in units of Hz, and the signal to noise ratio is improved at lower frequencies by including more sub-samples in the sample.

Possible values are “10”, “50”, “100”, “500”, “1000”, “5000”, and “25000”.

```
import requests
import json

# Device IP address
ip = "192.168.55.239"

def SetDataRate(rate):
    return requests.put("http://" + ip + "/t1/configuration/rate",
json.dumps(rate)).json()

# Set the data rate
SetDataRate("1000")
```

Similarly, the `SetDataRate` function sets the data rate of the T1 device. It sends an HTTP PUT request to the device's data rate endpoint. We use `json.dumps()` to convert the rate value to a JSON formatted string for transmission. The rate is passed as an argument when the function is called. In this case, we set the rate to "1000".

7.6 Collect full data rate field data and write to CSV using Python and WebSockets

This Python script collects data at the full data rate and writes it to a CSV file using WebSockets. It starts by initializing the IP address of the device, the amount of time to collect data, and the output file path.

Then it creates a database to store collected data and initializes a WebSocket connection to the device. The `sendEventData()` function sends the device an event structure that optionally contains a payload called data. The `sendSubscribeEvent()` function subscribes to the IO fields that are of interest. The `sendGetEvent()` function requests the device to send new data it has collected since the last time the function was called.

The `onMessageEvent()` function handles the response event from the device and processes newly collected data. It checks if the response is an update event that carries subscription data. If it is, the function appends the new values to the local database and sends another `sendGetEvent()` to request more data.

The script sends an initial subscription event and get event to start the collection process. It remembers the start time and waits for responses from the device. Once the collection time is over, the collected data is processed by writing it to a CSV file.

Finally, the WebSocket connection is closed, and the number of collected samples is printed to the console.

This example demonstrates how to collect data and write it to a CSV file using the WebSocket protocol. It can be used as a starting point for developing more complex data collection scripts.

```python
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()
```

8 Best Practices

1. **Use Parameterized IO Paths**: Always parameterize the IO paths to ensure that your code remains flexible in the event that the paths change with firmware updates. This will save you time in the long run as you won't have to make changes to your code each time a firmware update is released.
2. **Reuse Connections**: Reusing your sockets is highly recommended to make multiple requests. The firmware is designed to support recycling TCP connections for HTTP and WebSockets. Reusing connections will improve performance and save resources.
3. **Create Your Own Wrapper Functions**: Creating custom wrappers for the IO can be beneficial as it provides convenience for your specific application. All IO behaves the same way, so creating generalized functions can save you time and improve readability.
4. **Decouple from Specific Protocols**: HTTP may work well for your needs currently, but in the future, you may want to use EPICS instead. Writing your code in such a way that makes it easy to switch between protocols can save you time and effort.
5. **Ask for Help**: Pyramid is there to help you. If you have any questions or feedback on the interfaces, reach out to support@ptcusa.com. Your feedback can shape the development of the interfaces in the future.