

SCADA - User Manual

Author: Connor Winiarczyk

Introduction and Purpose

SCADA is an acronym for Supervisory Control And Data Acquisition. It is a term borrowed from industrial control applications, where a single server is often used to oversee large industrial plants such as oil refineries and assembly lines. In general, SCADA systems have three functions, they acquire data from the network of sensors connected to the plant, send control signals to the plant's other subsystems, and provide an interface for humans to interact with the plant by viewing aggregated data and issuing commands.

The Lafayette FSAE team has been working to develop a SCADA system that can be fully integrated into its electric vehicle, with the goal of performing all of the above three functions both during normal operation and throughout the various testing and maintenance procedures that it will undergo. At times in the past, this system has been referred to as VSCADA, short for Vehicle SCADA, to distinguish it from the industrial systems described above, but for brevity this document will refer to it simply as SCADA.

This document is referred to as a user manual, but could perhaps be better described as an inheritance manual. It includes information about how to use the software, but it also includes a bunch of other information that wouldn't be useful to an end user, but would be to someone inheriting this project with the intention of expanding upon or improving it. It includes information such as the overall design and structure of the project, why it was designed the way that it was, what I was thinking while designing it, and my personal thoughts for how it can be improved and expanded on. For that reason, it is more verbose than an ordinary user manual might be, but I hope that the reader will find this information useful.

Design Overview

CAN Network

The existence of a SCADA system implies a network. There must be communication taking place between SCADA and the other subsystems for it to successfully perform any of its responsibilities. There are a number of network protocols to choose from, but the de facto standard for automotive use is CAN (Controller Area Network) and so that is what is used in the Lafayette FSAE Car.

A good explanation of CAN can be found here:

<https://www.csselectronics.com/screen/page/simple-intro-to-can-bus/language/en>

At the time of writing, the CAN Network contains between 6 and 7 nodes, depending whether an external pc is needed to configure and operate the motor controller. They are:

- Motor Controller
- SCADA
- Battery Packs (1 and 2)
- TSI
- DashMan
- External Configuration PC

CANOpen

CAN is a very low level protocol. It defines a way for a node to broadcast up to 8 bytes of data and an ID, but very few of the other things needed to perform sophisticated network operations. For this, a higher level protocol needs to be defined on top of CAN, and currently there are several competing standards. These have been described as the equivalent of something like http to the tcp/ip stack, adding an additional layer of abstraction for easier use.

While it is not the most popular, the Lafayette FSAE team has chosen the CANOpen Standard for its vehicle in order to match that of the already purchased Motor Controller, which includes a rich set of CANOpen based tools.

CANOpen can also be thought of as a subset of CAN, meaning if a node is added to the network that does not comply with the CANOpen standard, the behavior of the other nodes is undefined. For this reason, it is very important that every node added to the network, even if it does not intend to make use of the full set of CANOpen features, at least comply with a subset of the protocol.

Here are some good resources for learning more about the protocol:

<https://www.can-cia.org/canopen/>

<https://www.youtube.com/watch?v=DlBkWryzJgg>

As a supplement to these resources, a brief description of CANOpen is also provided here:

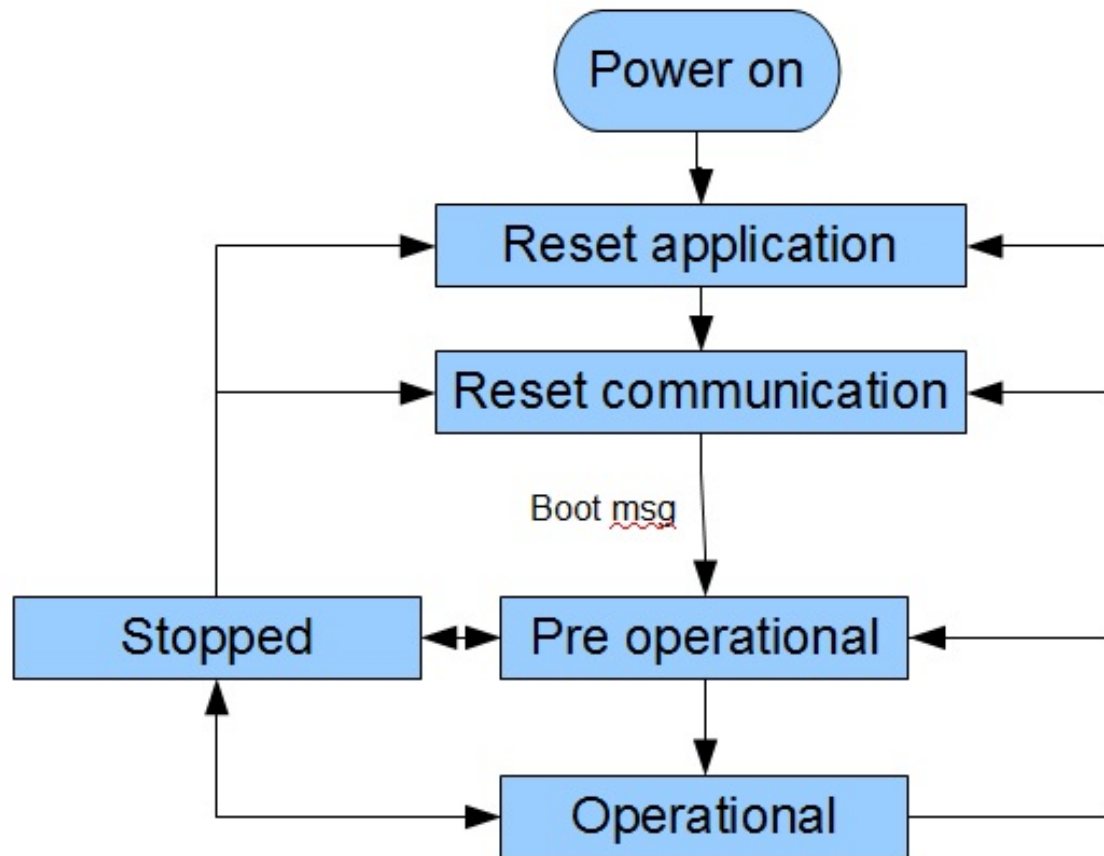
CANOpen can best be thought of as the sum of several communication protocols, coupled with some defined behavior for each node. Each has a different purpose, name, and associate acronym. A node can choose to implement any subset of these protocols and behaviors.

The protocol being used is determined by the function code, which is transmitted in the id part of the CAN frame. In the CANOpen standard, the message id is formed by the sum of the function code and the node id, this is to take advantage of the CAN behavior of prioritizing messages with lower ids, so protocols that are more important are given lower function codes.

NMT

NMT stands for Network Management. It is a master slave protocol used to manage the state of the various nodes on the

bus. Any node which wishes to interact with the NMT protocol as a slave must implement the following finite state machine.



This determines the functional state of the node. CAN packets can be sent by the master to move the node along this state machine, allowing it to perform actions, like soft resets, emergency stops, and boot ups.

SDO and OD

Any node which wishes to expose internal data to the CAN network must implement a data structure known as an Object Dictionary, or OD. The Object Dictionary maps each piece of data to an address consisting of a two byte index followed by a one byte subindex. Certain addresses are reserved for general data like device name and error registers. The manufacturer of a node can publish information about the Object Dictionary in a file called an Electronic Data Sheet, or EDS, which takes the form of an INI file that is both human and machine readable.

Any node on the network can access information from the Object Dictionary of another node using the Service Data Object, or SDO protocol. The SDO packet consists of a byte of metadata followed by a three byte address and 4 bytes of data. This can be used to both read and write data, and can be used to control node behavior in real time. This is the technique used in the dyno room to spin the motor and query it for data like temperature and angular velocity.

PDO

Process Data Objects are a protocol meant to supplement SDO's by providing a data transfer method with a higher data rate and less overhead. They are meant to be the standard for high volume inter node data transfer during nominal operation of the network, and have the added benefit of being much simpler to implement than SDO. A node that chooses to implement only the smallest possible subset of the CANOpen protocol will most likely implement a PDO.

PDO's work on a producer / consumer, or broadcast / subscribe model of communication, where one or more CAN packets

are sent at regular intervals, each containing 8 bytes of data with a structure agreed upon beforehand. Any node on the network can subscribe to these packets and update their behavior accordingly. Both the broadcast and subscribe behavior (called the Transmit PDO and Receive PDO respectively) can be configured by dedicated addresses in the Object Dictionary.

Hardware

SCADA runs on a Raspberry Pi that is mounted inside of the CarMan enclosure. Attached to it is a “hat” that enables CAN communication and provides a pass through for the Pi’s GPIO pins. Together, these are connected to the GLV board via a set of 3 cables: A micro USB cable for power, DB9 serial cable for CAN high and low signals, and a homemade ribbon cable for certain GPIO pins. Additional cables can be run to connect the Pi to other parts of the car and/or Dyno Room. These include:

- Ribbon Cable to the CarMan Display
- HDMI to to the Dyno Room Monitor
- USB to external keyboard and mouse
- USB to the dynamometer

author’s note

The three cables connecting the SCADA Pi to the GLV board are a bad design decision and should be removed at the next possible opportunity. With the exception of the DB9, these cables and their connectors are not rated for automotive use, and could easily become disconnected during heavy vibrations and render SCADA inoperable. In addition, the adjacency of the SCADA Pi and the GLV Board within the CarMan enclosure make these cables appear comically large, forcing space to be used for cable routing that could otherwise be taken up by something more useful. It would be a far better solution to integrate the SCADA Pi and the CAN hat into the GLV board.

Raspberry Pi offers a compute module, which is the computer as the ordinary Raspberry Pi 3, but with some peripherals removed, and condensed into a much smaller form factor with a single card edge connector as its only interface. I highly recommend that future teams replace the existing SCADA Pi with one of these, with external components like the CAN interface and ethernet jack integrated into the GLV board.

<https://www.raspberrypi.org/products/compute-module-3-plus/>

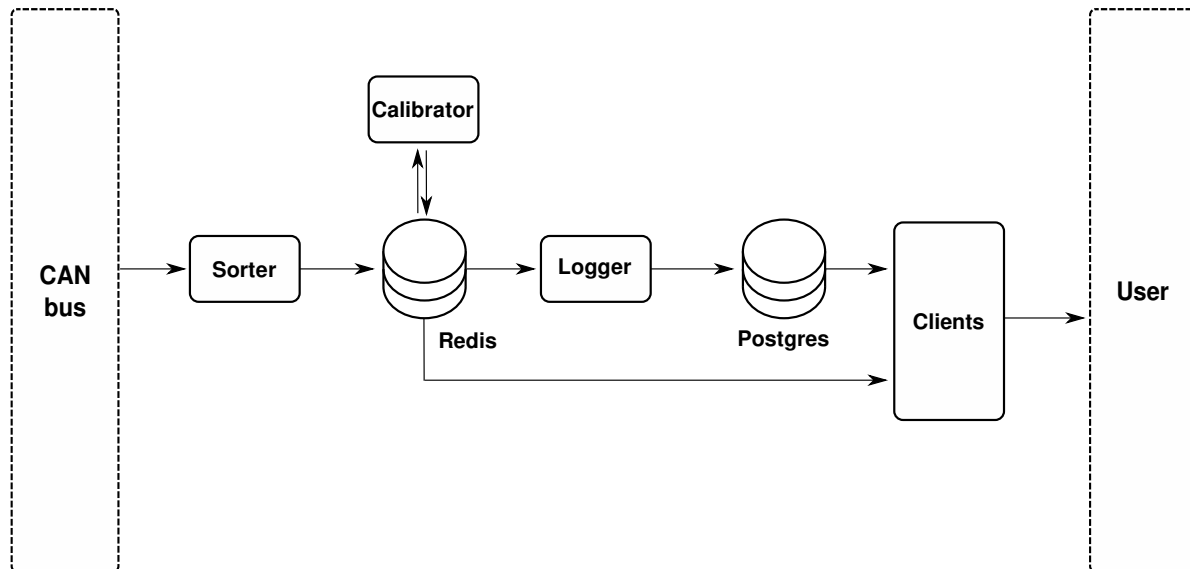
Software

The most important thing to understand about the design of SCADA in its current form is that it is not a single program, but rather a system of programs which are made to run concurrently and interface with each other. These programs try as much as possible to abide by the unix philosophy of software development, which is in essence, to write programs that are extremely limited in scope, and that interact well with each other, through standard, well defined interfaces. This is opposed to the way that SCADA has been written in the past, where a single large, integrated program is made to implement all the features required by the system. There are some drawbacks to the small and modular approach, which will be addressed later, but also a lot of advantages. They include the following:

- Features can usually be added without having to modify existing code, this can help to mitigate the square law of adding features to things.
- SCADA can be written in multiple different languages, so the best language can always be chosen for a given purpose, without having to make compromises.
- When the purpose of a program is general enough, existing open source software can be used instead of writing an in house solution.
- The individual components of the system are easy to understand and reason about.
- When problems arise, they are easy to isolate and diagnose.

The programs making up SCADA and their structure are described in the following diagram. They are organized into a pipeline, with data flowing from one service to the next in a well defined way. Each service has a specific roll in either

organizing or transforming the data so that it can be made sense of by the user.



SCADA Data Aquisition Pipeline

Data flows from the CAN interface to the user through a series of services and storage mediums. Each of which serves a distinct purpose.

What follows is a short description of each service and its function.

Sorter

The sorter listens to the CAN bus for incoming messages, upon receiving a message, it checks its ID and Structure against a config file to generate an associated set of key value pairs. It then writes these key value pairs into the Redis cache and sends a message to the calibrator that new data has been received. The sorter is the only service that is aware of CAN or CANOpen, meaning SCADA can be made to work with a different networking protocol, or more than one, simply by switching out the sorter service or adding an additional one, without affecting any services downstream.

Calibrator

The calibrator is responsible for translating the raw data read by the sorter from the CAN bus into a more human readable form. This is usually a case of linear calibration of raw sensor data, such as with the TSI node, or the recombining of data that was transmitted as multiple bytes, like with the Throttle value reported by Motor Controller. However, it can be made to support calculations of arbitrary complexity, such as for example, the calculation of power from a given voltage and current, the averaging of multiple sensors, filtering, differentiation, and so on. All calibrator operations are defined by the user as functions in the user_cal.py file. Once data is calculated, it is written back into the Redis server as a different key.

Logger

The logger takes the data gathered by the sorter and calculated by the calibrator and logs a subset of it into the Postgresql database at regular intervals. The subset of data to be logged is defined in config.yaml file.

It samples the data at regular intervals, and performs some basic logic to limit the database to only storing useful data. In general, it tries to only log data that is changing, so as not to flood the database with constant values. The logger writes all logged data to a single sql table, called `data`, with the following columns:

- `id`
- `sensor_id`
- `value`
- `timestamp`

In keeping with the unix tradition, all data is stored in the same way as text. Programs reading from the database are responsible for translating the data into its expected type. A companion table, called `sensors` is included in the database to store metadata about each sensor. It has columns:

- `id`
- `redis_key` (equivalent to `sensor_id`, one of these should probably be renamed)
- `display_name`
- `datatype`
- `unit`

Redis

For the most part, programs communicate with each other via Redis. Redis is an open source, in memory database that stores data as key value pairs, it also functions as a basic communications bus for sending simple inter-service messages.

<https://redis.io/>

Postgres

If a program needs to write non-volatile data, it does so using an SQL database via a Postgres server.

<https://www.postgresql.org/>

Concurrency

Concurrency is handled by the operating system via `systemd`. All programs which are meant to be run as a service have an associated `.service` file, and can be managed with the `systemctl` interface. All services are single threaded and should consist of only one update loop, delay statements are inserted into the loop in order to release the cpu for use by other services. A good example of an update loop for a service that responds to redis messages is included below:

```
while True:
    message = p.get_message()
    if message:
        update()
    else:
        time.sleep(0.1)
```

<https://www.digitalocean.com/community/tutorials/how-to-use-systemctl-to-manage-systemd-services-and-units>

Clients

Both the Redis and Postgres servers expose TCP ports, and any software capable of interacting with these TCP ports can be considered a SCADA client. These clients can be run either locally on the SCADA Pi or remotely over a network. There are already a number of SCADA clients to choose from, and the best one for any given application will depend largely on circumstance and preference.

One potential 3rd party client that comes very highly recommended is Grafana. Grafana is an open source data viewing and monitoring tool that specializes in the construction of a wide range of data visualizations such as graphs, gauges, statistics, and logs based on a number of potential data sources. It supports Postgresql out of the box and is extremely easy to set up. It also has a rich community of plugin developers.

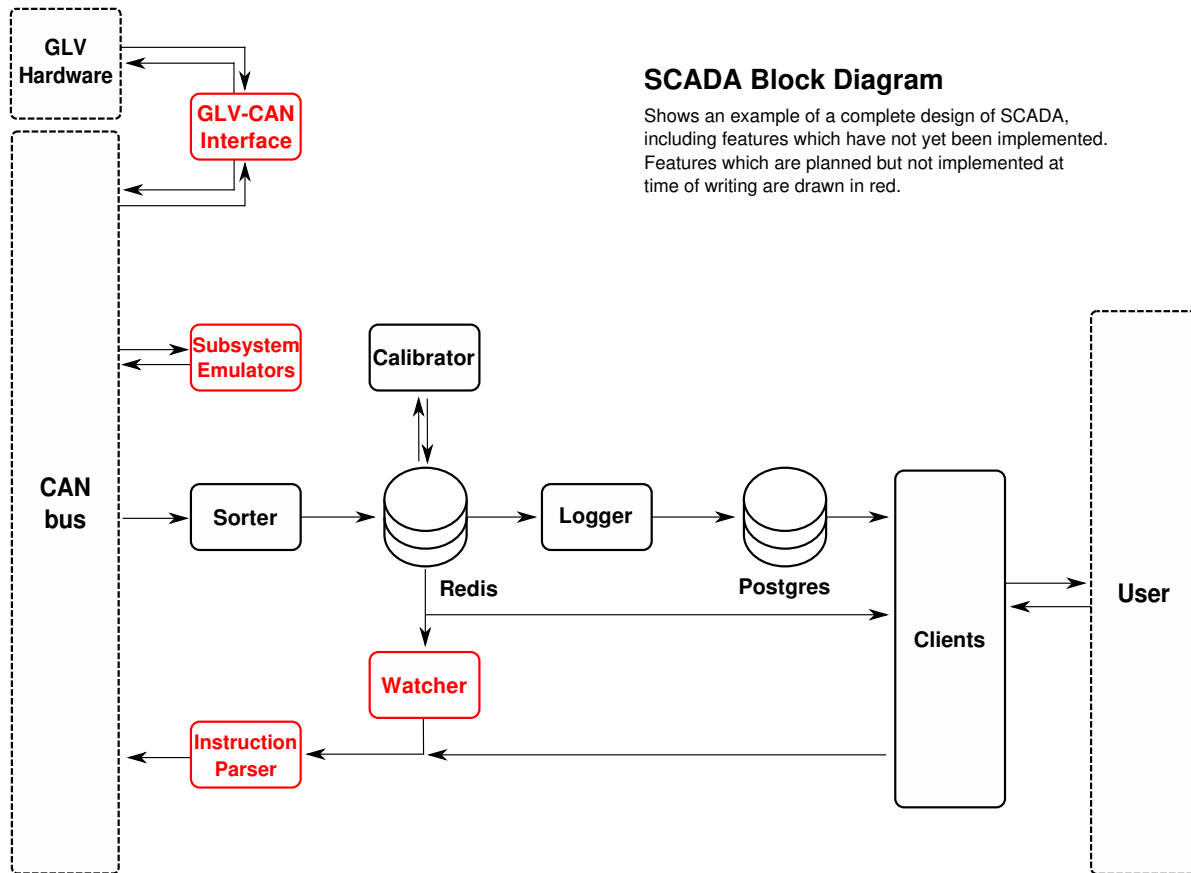
<https://grafana.com/>

Other clients are in development as well, including a command line tool for basic management, a curses based monitoring tool for easy testing over ssh, a Java based tool for graph generation, and a Tkinter based tool for graphical live monitoring and management.

Future Goals

At the moment, SCADA is a strictly passive system in its relationship to the CAN bus. The services described above implement a pure data acquisition system which listens to the CAN bus and presents the incoming data to the user, but does not have the ability to talk on that bus to issue commands or configure the network for certain behaviors. This means that SCADA is, by definition, incomplete. There needs to be a way to perform supervisory control functions by talking back to the network.

Because data acquisition is the most important part of the FSAE team's need for a SCADA system, these were the services that were built first, but more work is required to turn this into a complete SCADA system. In addition, the SCADA PI needs to perform some secondary services that are not strictly part of the SCADA system. An example of a more complete SCADA design that includes non-implemented features is shown below.



A list of the additional services depicted and a description of their function is included below:

Instruction Parser

The instruction parser would serve as a counterpart to the sorter. Instead of reading from the CAN bus, it would write to it, and serve as the bridge through which the rest of SCADA could send control signals to the car's other subsystems. In order

to keep the other SCADA services agnostic about exact CAN message syntax, a text based format should be agreed upon for representing control signals that can sent over the CAN network. The instruction parser would then have the sole responsibility of parsing this format into CAN messages.

A proposal for such an instruction set is included with SCADA documentation and can be read here:

<https://raw.githubusercontent.com/Lafayette-FSAE/scada/master/documentation/sparkyscript.pdf>

Watcher

The instruction parser would, by default, receive most of its instructions from client programs, which would in turn receive instructions from the User, who would make decisions about which commands to send by viewing the incoming data. This forms a feedback loop for doing certain kinds manual system control. However, there will most likely be a need for an additional, automatic feedback loop which can take place at a higher frequency and without user intervention.

In theory, a client could be written to perform such a task, but it would not continue to run after it was disconnected. Instead, a dedicated service should be written, which regularly polls the Redis server for the state of the car, and makes decisions about what commands to send based on this data. It could be configured as a list of condition, instruction pairs, and would be useful for things like opening the safety loop during emergency conditions.

GLV-CAN Interface

The GLV board contains a number of sensors and control vectors that should ideally be exposed to the rest of the car using the standard CANOpen interface. This could, in theory, be done by a dedicated microcontroller, like is done with the car's other subsystems, but the proximity of the SCADA Pi to these sensors has meant that, for better or for worse, it has absorbed these responsibilities.

Nevertheless, the GLV-CAN interface should be thought of as distinct from SCADA, and written as its own program that only interfaces with SCADA through the CAN bus. This has the drawback of requiring some extra steps in the communication process, but it allows other nodes on the network to access data from the GLV system. In addition, it helps to keep SCADA implementation more "pure" in the sense that it does not require the need for built in exceptions and special behavior based on the method the data was obtained. Previous implementations of SCADA were riddled with such exceptions, and they were a common source of complaints.

Subsystem Emulators

In the absence of a fully integrated system, testing the CAN interface of a given subsystem can be a difficult task. For example, you may want one of the batteries to behave differently based on the state of the other one, but only have constructed a single prototype. The DashMan system may want to test the way it responds to a certain TSI state, but the last TSI board may have been fried do to a wiring error. And of course, you may be forced by unforeseen circumstances to develop SCADA and all other subsystems remotely and while forbidden to come within six feet of the lab, the car, or eachother.

For such events, the 2020 team has begun work on a series of Subsystem Emulators, which are meant to be run either on the SCADA Pi or a separate CAN connected computer, and mimic the CAN interface of each subsystem. Each emulator should mimic its subsystem's Object Dictionary, SDO server, and Transmit PDO messages. While not necessary, additional hardware like LEDs and buzzers would also be fun.

Extra Ideas

SDOs

The instruction parser would also be required for reading SDO data from the network. This is because the SDO operates on the client server communication model, rather than the producer consumer model used by PDO data.

How to Use

Installation

Prerequisites

SCADA assumes it is being run on a debian based linux distribution. This will almost always be raspbian, but it has also been tested on a pure debian server. This should also work on something like ubuntu, but this has not been tested. SCADA will not work on a Windows or Mac, and most likely never will. This is by design, as SCADA takes advantage of a large number of assumptions about its environment.

It is recommended to set up a dedicated development server for SCADA using a spare raspberry pi and ideally something resembling the GLV hardware. This could be expanded over time into include a mock CAN bus for integration testing with other subsystems.

Because the Lafayette network can be difficult to navigate, it is recommended that this be bypassed using either a dedicated physical network or vpn. This could be set up in such a way as to prevent ip address changes and to enable off site work.

Install Script

One of the drawbacks of the small and modular approach to system development is that installation becomes a bit more tricky. To help mitigate this, an install script is included with SCADA which is meant to help automate the process.

Ideally, a full installation of SCADA could be performed with the following shell commands:

```
$ git clone https://github.com/Lafayette-FSAE/scada
$ cd scada
$ sudo bash install
```

However, because SCADA is still relatively new software, the install script is likely to fail in some environments, requiring manual intervention. In addition, updates will need to be made to the install script periodically as new features are added to SCADA. To that end, an explanation of the install script and what it does is provided here.

```
apt-get install python3
apt-get install python3-pip
apt-get install redis-server
apt-get install can-utils
```

This section uses apt-get to install 3rd party programs and dependencies. New dependencies can be added to SCADA by appending apt-get calls to this list.

```
# install python dependencies
pip3 install python-can
pip3 install redis
pip3 install blessed
pip3 install pycopg2-binary
```

Next, the python package manager is used to install python specific dependencies.

```
# make sure virtual can bus is set up for testing
modprobe vcan
ip link add dev vcan0 type vcan
ip link set up vcan0
```

The Pi's CAN interface is created and turned on. This is currently configured to use a virtual CAN interface for development and testing, but can be switched to the real CAN bus by replacing all instances of vcan with can.

```
# make binary files executable
chmod +x install
chmod +x scada
chmod +x sorter/sorter.py
```

```
chmod +x calibrator/calibrator.py
chmod +x logger/logger.py
```

Files which need to be executable are made so explicitly with the `chmod` command. This includes scripts which are run as services, the `scada` cli interface and this install script.

```
# copy binary files to /usr/bin
cp scada /usr/bin/scada
cp sorter/sorter.py /usr/bin/scada_sorter.py
cp calibrator/calibrator.py /usr/bin/scada_calibrator.py
cp logger/logger.py /usr/bin/scada_logger.py
```

Copy executable files into a known directory. The exact directory chosen is sort of arbitrary, so long as it matches the directory written in the `.service` files. `/usr/bin` is a good choice because it is the standard for user installed binary files and it is already in the `PATH` variable.

```
# create a workspace and copy important files into it
mkdir -p /usr/etc/scada
rm -rf /usr/etc/scada/Utils
cp -r Utils /usr/etc/scada/Utils
rm -rf /usr/etc/scada/config
cp -r config /usr/etc/scada/config
cp ./install /usr/etc/scada
```

This also copies files into a known directory, this time the non executable files, and to the `/usr/etc/scada` directory. Again this was chosen arbitrarily. This is where the services will look for things like custom python libraries and config files.

```
# copy service files for systemd
cp sorter/sorter.service /etc/systemd/system
cp calibrator/calibrator.service /etc/systemd/system
cp logger/logger.service /etc/systemd/system
```

Finally copy the `.service` files into a place where `systemd` can find them. This directory is not arbitrary, and must be `/etc/systemd/system`

Verification and Troubleshooting

If SCADA was installed correctly, you should now have a command line helper tool that can be used for basic system management. Verify this by typing `scada --help` into the command line and running it. You should see a help message describing the possible sub commands that can be used.

The status of each service can be verified using the command `scada status`, which is equivalent to running `systemctl status <service>` for each service included in `scada`. You should see an entry for `calibrator.service`, `logger.service`, and `sorter.service`, if all is working they should all be green and have a status of `active`. If one or more service is missing, they were probably not installed correctly, you can look for their service files in the directory `/etc/systemd/system`.

If one or more of these services is in the failed state, that usually indicates that there was an error running their associated python script. These errors can usually be found by checking the logs, which are accessed with the command `sudo scada logs`.

Services can also be troubleshooted by stopping them and running the script manually. The command `sudo scada stop <service>` will stop a given service, and `sudo scada start <service>` can be used to restart it. Any service can be run manually by invoking its python script file directly. For example `python3 ~/scada/sorter/sorter.py`

If all services are present and running without errors you should begin testing the system against some real can data. If no external can bus is available, the included subsystem emulators will generally be the easiest way. These do not have associated `.service` files and must be run manually. Navigate to the `~/scada/emulator` directory and run `python3 main.py`. For convenience, it is best to run this in a `tmux` window so that other tasks can be performed with this program in the background.

This is a very good guide about how to set up and use `tmux`. <https://www.hamvocke.com/blog/a-quick-and-easy-guide-to-tmux/>

Once the emulator is up and running, you should test to see if it is generating CAN traffic. The best way to do this is by using the `candump` command that is built in to `can-utils`. Type the command `candump vcan0` and run it, you should see a regular stream of can packets. If the TSI emulator is running, you will see a regular packet with the can id of 183, this is the sum of the function code 0x180, indicating that it is a PDO packet, and the node id of 3, indicating that it is coming from the TSI. If you are using a different CAN interface, you will need to switch the `vcan0` argument to the correct one. This will be `can0` in the live system.

The sorter and calibrator can be verified by looking at the Redis cache directly. Type the command `redis-cli --scan` to get a printout of all active keys. Redis data written by the sorter and calibrator will expire after a period of 10 seconds, so a lack of data usually indicates that one or both of them have stopped. The value of one of these fields can be read using the command `redis-cli get <key>`. A good test is to query the `tsi:throttle` key, it should be changing over time.

The `redis-cli` is a very powerful tool, and is effectively its own SCADA client. More info about how to use the `redis-cli` tool can be found here:

<https://redis.io/topics/rediscli>

The last step is to verify that the logger is writing data into the database correctly. The easiest way to do this is by querying the database with the `psql` command.

Running the command

```
psql -h localhost -p 5432 -d <databasename> -U <username>
```

The `databasename` and `username` will be the ones chosen while setting up Postgres. This will prompt you for a password, which is also chosen during the setup process. If successful, you will be entered into a REPL where you can execute SQL commands against the database.

Typing `\dt` will show the list of tables, and should include at least a data and sensors table. Running the command `select * from data;` will show the contents of the data table. Confirm that the logger is working correctly by checking the end of the table to see if new data is being written.

Configuration

While SCADA consists of several distinct programs, for the sake of convenience, they all read from the same set of configuration files. At the moment, there are two, both of which can be found in the `config` directory.

`config.yaml` is a general purpose config file which handles the majority of options. `user_cal.py` is a python file specific to the calibrator which defines the calibration functions it will use to map data as it comes in. Together they make up the full set of configuration options for SCADA.

The following sections explain the configuration options in further detail.

Bus Info

The bus info section gives the sorter information about the CAN bus. If you are running SCADA on a system with a virtual CAN interface, set `channel` to `vcan0`. As it is currently configured, the raspberry pi in the Dyno room talks to the rest of the network on the `can0` channel of `socketcan` with a bitrate of 125000

```
bus_info:
  bustype: socketcan
  channel: can0 | vcan0 | (or any other socketcan channel)
  bitrate: 125000
```

Data

Most data is sent over the bus in 8 byte CAN packets called Process Data Objects, or PDOs, with each byte representing a different piece of from that node. The `process_data` fields tells SCADA where to expect each piece of data in each packet

```
process_data:
  MOTOR: [ STATUS, DUMMY2, TORQUE, DUMMY4, DUMMY5, DUMMY6, DUMMY7, DUMMY9 ]
```

```

MOTOR-2: [ MOTOR_TEMP, CONTROLLER_TEMP, DC_LINK_VOLTAGE, WARNING, CURRENT_DEMAND, TEST ]
MOTOR-3: [ THROTTLE-byte0, THROTTLE-byte1, AUX, BRAKE, PHASEB_CURRENT, DUMMY5, DUMMY6, DUMMY7 ]
MOTOR-4: [ TORQUE_REGULATOR, FLUX_REGULATOR, VELOCITY ]

TSI:    [ conditions, 'mc_voltage:raw', 'voltage:raw', 'cooling_temp:raw', 'throttle:raw',
          'flow_rate:raw', 'state:int', 'current:raw' ]
PACK1:  [ voltage, current, state_of_charge_01, state_of_charge_02, temp, 'cells:temp:min',
          'cells:temp:avg', 'cells:temp:max' ]
PACK2:  [ voltage, current, state_of_charge_01, state_of_charge_02, temp, 'cells:temp:min',
          'cells:temp:avg', 'cells:temp:max' ]

```

While not yet implemented, SCADA should eventually implement an SDO client for the ability to poll the network for additional data not included the devices PDOs. For that reason, a service data section is also defined to give instructions to this client about what data it needs to poll and how often.

The service_data section defines a list of data that needs be manually polled in this way. Each piece of data needs to have a node_id, index, subindex, and poll_rate

```

service_data:

Cell1Temp:
  node_id: 2
  index: 2011
  poll_rate: 10

MotorTemp:
  node_id: 1
  index: 2010
  subindex: 0
  poll_rate: 0.5

```

NOTE: it is important to remember that not all nodes on the bus will support this, but the Motor Controller definitely will.

A complete description of all data that can be accessed from the motor controller can be found [here](#)

Further reading on Service Data and Process Data can be found

<http://www.byteme.org.uk/canopenparent/canopen/sdo-service-data-objects-canopen/>

and

<http://www.byteme.org.uk/canopenparent/canopen/pdo-process-data-objects-canopen/>

Calibration

Calibration is configured in a separate python file called `user_cal.py` in the calibration folder. A Calibration function is defined using the `@cal_function` function decorator, which takes two arguments:

- output: the name of the data being generated
- arguments: a list of the data needed as arguments

Because the calibrator operates only on data within the Redis cache, `output` and `arguments` should both contain Redis keys. These can be any string in theory, but, by convention, consist of all lower case characters, and are organized into hierarchies via the `:` character. (For example, all keys that store data about the TSI take the form `tsi:*`)

It is important that the keys written in the arguments list correspond to those those written in the data section of the `config.yaml` file. Otherwise, the sorter and calibrator will not agree on where to look for data. It is also important to ensure the output key does not conflict with any other input or output keys.

To help avoid key collisions, when multiple keys are used to represent the same piece of data, but transformed in some way, such as in the case of a linear calibration or unit conversion, they should be distinguished from each other with the use of a second `:` followed by a modifier. Only the key that is meant to represent the default form of the data should omit this modifier. The grouping of keys in this way allows for easier querying of the available data by downstream programs. For example, in the case of the TSI, data is transmitted as raw sensor values, and converted into usable units by calibration

functions. The non calibrated data is stored with the key `tsi:<data>:raw` and its converted form is stored under `tsi:<data>`.

Examples

1. Celsius to Fahrenheit

```
# Converts ambient temp of pack1 to fahrenheit because
# we live in America god damn it
@cal_function(output='pack1:temp:fahrenheit', arguments=['pack1:temp'])
def packtemp_fahrenheit(args):
    temp, *other = args
    temp_fahrenheit = temp * (9/5) + 32
    return temp_fahrenheit
```

2. Voltage and Current to Power

```
# Calculates the TS power draw in kW
@cal_function(output='tsi:power', arguments=['tsi:voltage', 'tsi:current'])
def ts_power(args):
    voltage, current, *other = args
    power = (voltage * current) / 100
    return power
```

3. Motor Controller Voltage from Sensor Data

```
@cal_function(output='tsi:mc_voltage', arguments=['tsi:mc_voltage:raw'])
def mc_voltage(args):
    mc_voltage_raw, *other = args
    return ((mc_voltage_raw / 255) * 5) - 1.28) * 61
```

4. TSI state enumeration

```
@cal_function(output='tsi:state', arguments=['tsi:state:int'])
def state(args):
    state_number, *other = args

    if state_number == 1:
        return 'GLV-ON'
    elif state_number == 2:
        return 'AIRS-CLOSED'
    elif state_number == 3:
        return 'DRIVE SETUP'
    elif state_number == 4:
        return 'READY TO DRIVE SOUND'
    elif state_number == 5:
        return 'DRIVE'
    else:
        return 'STATE UNDEFINED'
```

5. Motor Controller Throttle

```
@cal_function(output='motor:throttle', arguments=[
    'motor:throttle:byte0',
    'motor:throttle:byte1'
])
def throttle(args):
    lsb, msb, *other = args
    return msb * 256 + lsb
```

Data generated by these functions will be written to the same data cache structure as the rest, and will have an index determined by the `output` argument, allowing it to be accessed by other programs downstream.

Viewing Data

SCADA data can be viewed by any SCADA client. A client is defined here as any program meant to interface with the

SCADA services through the TCP ports exposed by Redis and Postgresql. For reference, these are by default port 6379 and 5432, respectively. As seen in some of the previous sections, command line tools provided by both Redis and Postgres can be used as crude SCADA clients, and are useful for quick debugging. However, most people will want to use something more visual.

In order to support at a glance system monitoring over ssh, a curses based client is under development and included with `scada`. It can be run with the command `scada tui`. Unfortunately, it is not yet considered complete. There is a lot of potential for improvement, and could prove quite powerful, but like all user interfaces, it is mainly a matter of preference, and future teams should develop the client they find most useful.

Grafana

Grafana is an open source tool for generating live graphs and other data visualizations with an emphasis on system monitoring and diagnostics. It was built primarily for the web development community as a frontend for timeseries databases and monitoring tools like prometheus and graphite, but it is flexible enough to be used for a wide range of applications, such as SCADA systems.

The best way to get a sense for what grafana is how to use it is to try out their playground server, found here:

<https://play.grafana.org>

You can also read more about the project here: <https://grafana.com/>

or here: <https://github.com/grafana/grafana>

Setting up Grafana

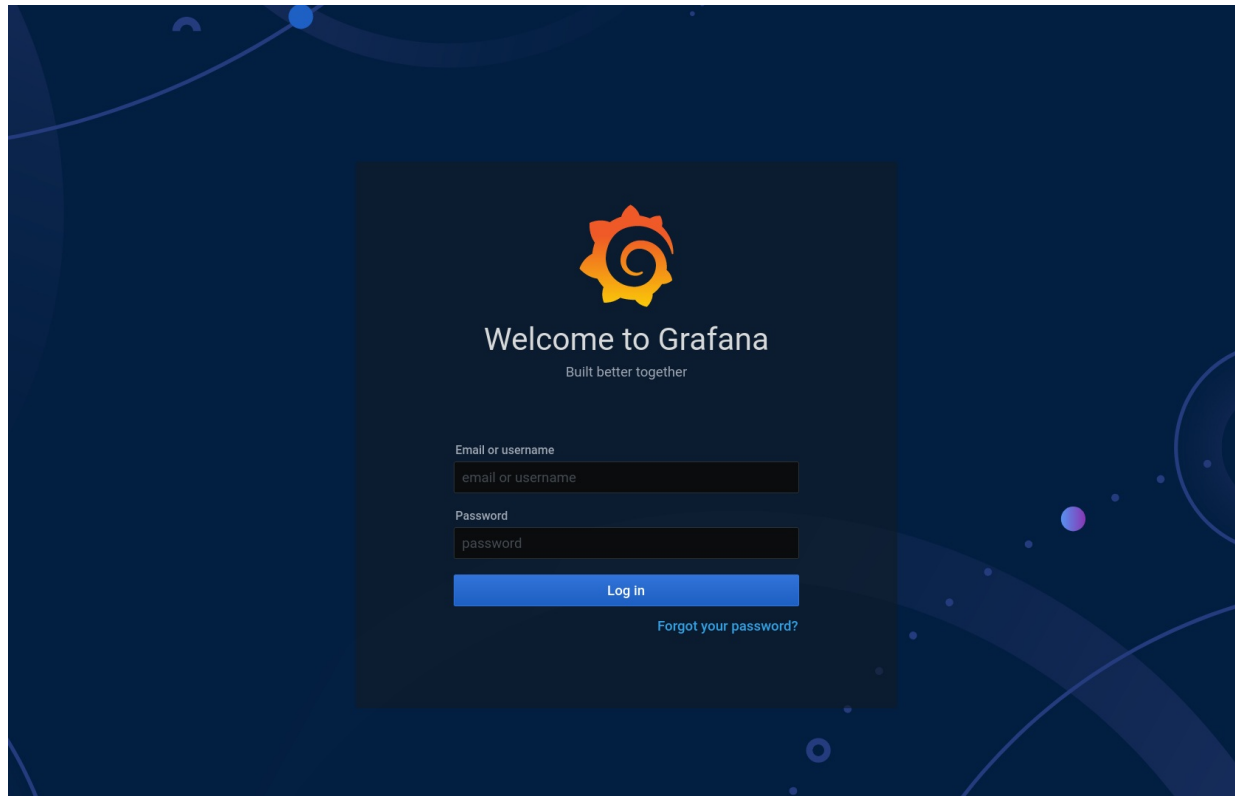
Grafana runs as its own server, so it can be installed anywhere with an internet connection, and most likely shouldn't be installed on the same Raspberry Pi as SCADA. There are a number of ways that a Grafana server can be installed, including through the apt-get repository or into a docker container via the official docker image:

<https://grafana.com/docs/grafana/latest/installation/docker/>

Once installed log into the grafana server by navigating to the ip address of the device you installed it on with the default port of 3000.


Logging In

The login screen will look something like this. Enter the default username and password of admin, admin.




Adding a Data Source

Click the configuration icon on the left hand side of the screen. Go to data sources, and click the add data source button on the right. Choose PostgreSQL from the list.

 **Add data source**
Choose a data source type


Cancel



PostgreSQL
Data source for PostgreSQL and compatible databases
Core

Select

Fill out the form with the appropriate info. The host should be the ip address of the computer SCADA is running on, followed by the port 5432. The rest will have been info chosen when setting up the database. In my case, I use the database “demo” with the user fsae and password cables. Be sure to set SSL Mode to disable.

 **Data Sources / SCADA Database**
Type: PostgreSQL

Settings

Name ⓘ SCADA Database Default ☐

PostgreSQL Connection

Host54.226.155.95:5432

Databasedemo

UserfsaePasswordconfiguredReset

SSL Modedisable ⓘ

Connection limits

Max openunlimited ⓘ

Max idle2 ⓘ

Max lifetime14400 ⓘ

PostgreSQL details

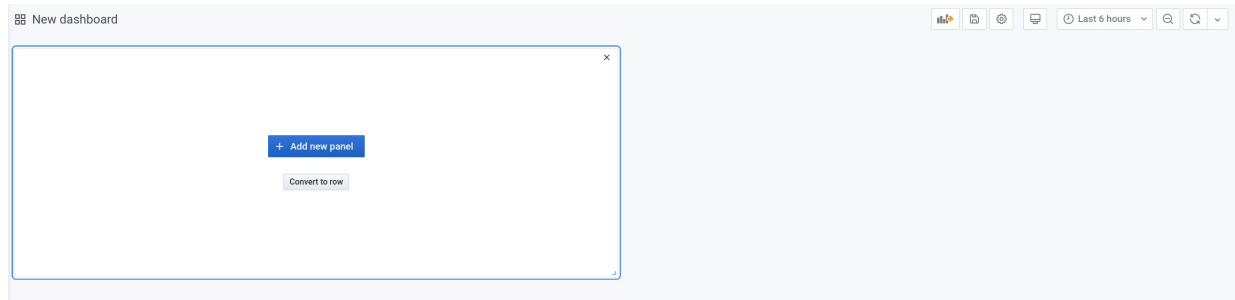
Version ⓘ9.3 ▾

TimescaleDB ☐ [Help >](#)

Min time interval1m ⓘ

Creating a Panel

Once the database has been added, you can begin creating dashboards. On the grafana homescreen, click the plus button on the left hand side of the screen and select create dashboard. Once created, you can add to the dashboard any number of panels. Each panel combines an SQL query with a number of additional settings to create one of several visualizations. Some examples are included below:



Drive State

A drive state indicator can be made from the STAT visualization. The following SQL query is used to get the most recent entry of the drive state sensor in the data table:

```
SELECT
  value
FROM
  data
WHERE
  sensor_id='tsi:state'
ORDER BY timestamp desc
LIMIT 1
```

In the settings tab, set calculation to last and fields to all fields. The resulting panel should be a string describing the most recently logged Drive State. This can be verified by restarting the TSI emulator or board, and taking it through the start up sequence.

The screenshot shows the Grafana dashboard editor interface. At the top, there's a navigation bar with 'New dashboard / Edit Panel' and buttons for 'Discard', 'Save', and 'Apply'. Below this is a toolbar with 'Fill', 'Fit', 'Exact' options, a time range selector set to 'Last 6 hours', and search, refresh, and zoom icons. The main panel area displays a visualization titled 'TSI State' showing the word 'DRIVE' in large green letters. Below the visualization is a query editor with the following SQL query:

```
SELECT
  value
FROM data
WHERE
  sensor_id='tsi:state'
ORDER BY timestamp
LIMIT 1
```

Below the query editor are tabs for 'Format as' (set to 'Table'), 'Query Builder', 'Show Help', and 'Generated SQL'. To the right of the main panel is a settings sidebar with the following sections:

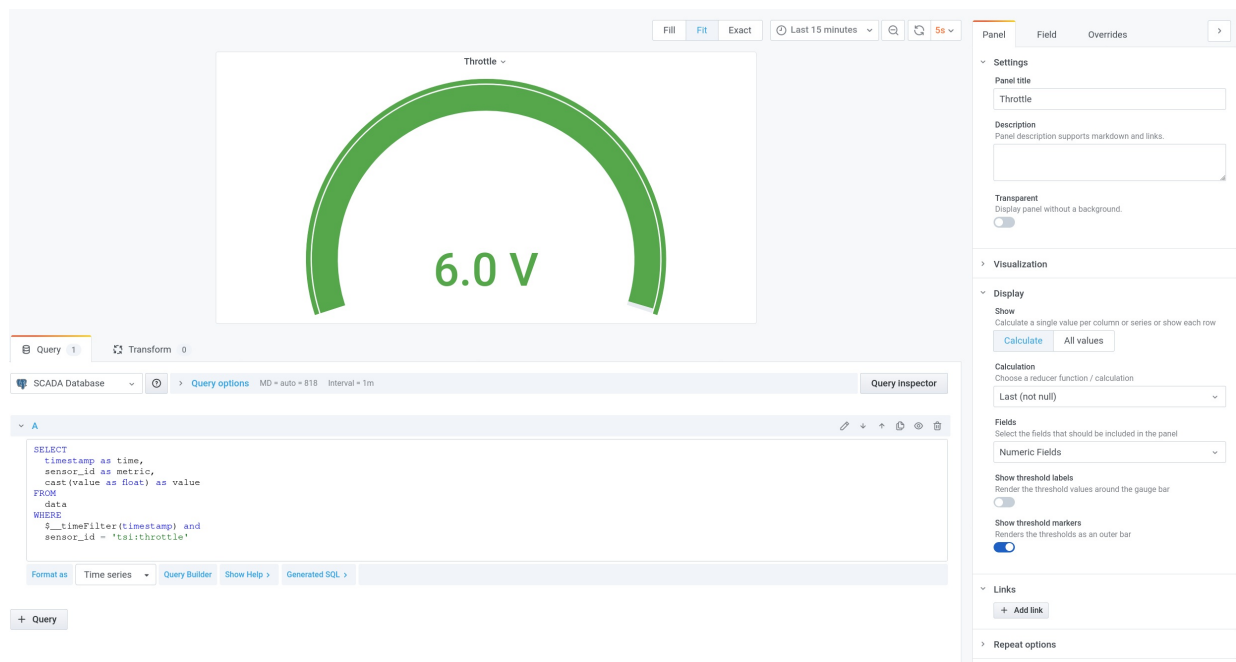
- Settings**
 - Panel title: TSI State
 - Description: Panel description supports markdown and links.
 - Transparent: ☐ Display panel without a background.
- Visualization**
 - Display**
 - Show: Calculate a single value per column or series or show each row. Buttons: Calculate, All values.
 - Calculation: Choose a reducer function / calculation. Dropdown: Last.
 - Fields: Select the fields that should be included in the panel. Dropdown: All Fields.
 - Orientation: Stacking direction in case of multiple series or fields. Buttons: Auto, Horizontal, Vertical.
 - Text mode: Control if name and value is displayed or just name. Dropdown: Value.
 - Color mode: Color either the value or the background. Buttons: Value, Background.
 - Graph mode: Stat panel graph / sparkline mode. Buttons: None, Area.

Throttle Value

The gauge panel is a great way to visualize numerical data, like throttle value or pack state of charge. The following SQL is used to query for the most recently logged throttle data

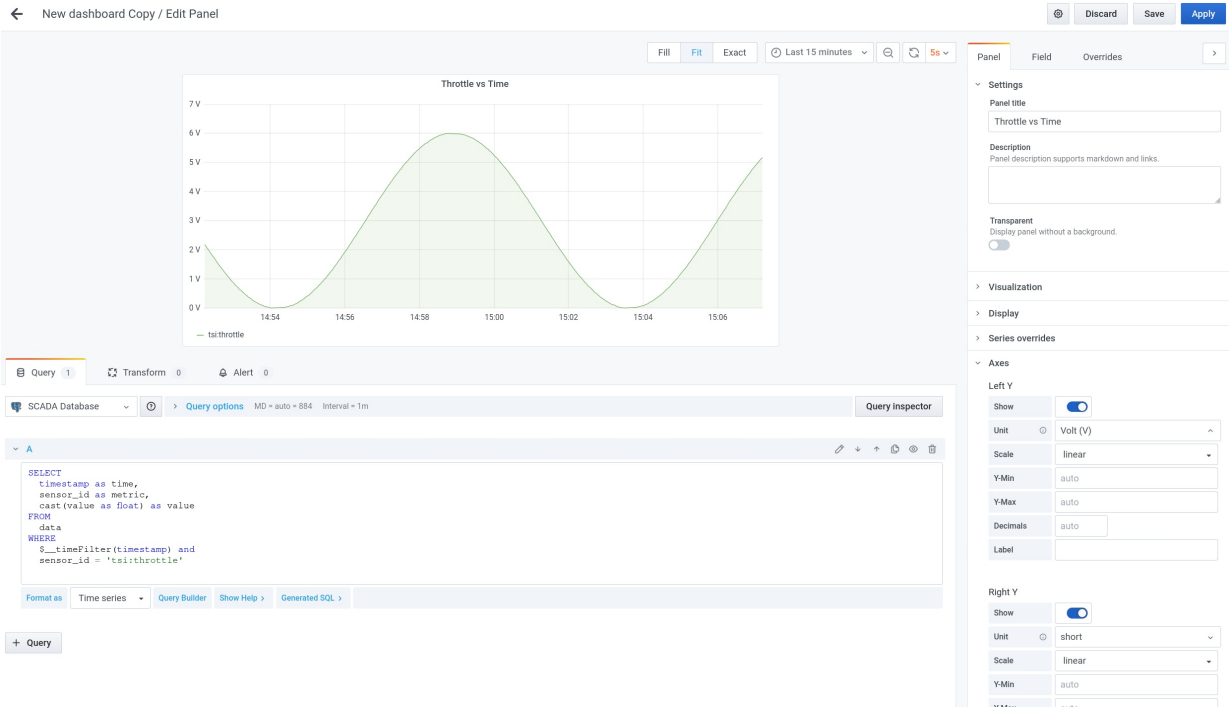
```
SELECT
  timestamp as time,
  sensor_id as metric,
  cast(value as float) as value
FROM
  data
WHERE
  $__timeFilter(timestamp) and
  sensor_id = 'tsi:throttle'
```

Note that we have to explicitly cast the value to a float here, because otherwise it will be treated as a string. Again, calculation should be set to last.



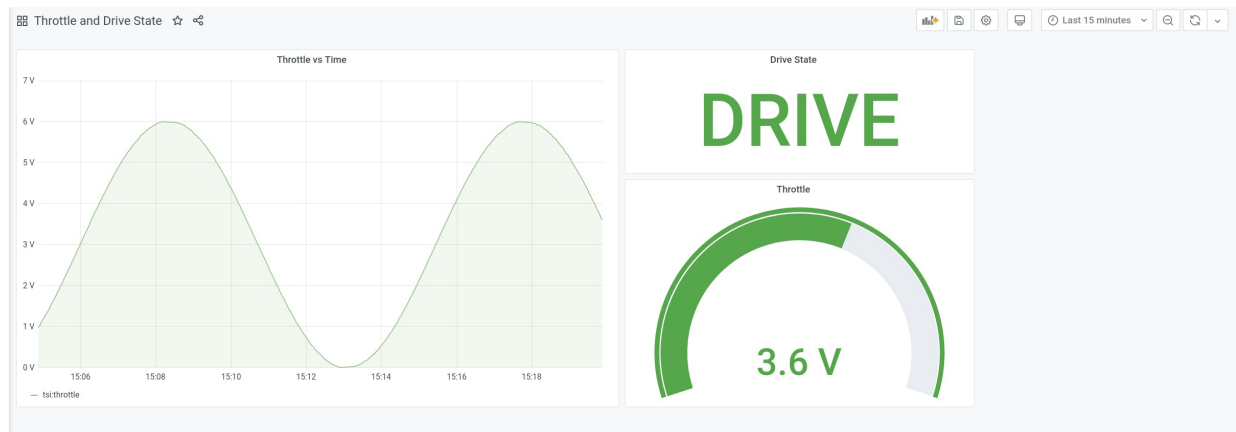
Throttle History

The same query can also be used to graph a history of throttle values over time, simply by choosing the graph visualizer instead of the gauge. Here we can see that the throttle value is following a sinusoid, which matches what we expect based on the mock behavior of the TSI board.



Final Dashboard

The result of all these panels will be a dashboard that looks something like this. We now can log in to the grafana server and see a live display of the car's drive state, its current throttle input as a voltage, and the history of that input over any given timerange. As the team adds more sensors and CAN nodes, it can seamlessly add additional panels for viewing their data in whichever way is most appropriate.



author's note

Grafana is awesome. It is flexible, easy to use, and can be integrated into the existing SCADA system for free and at no cost to performance or complexity. Frankly I regret not discovering it sooner, since it would have provided an easy answer to most of the questions we had while SCADA was being developed. I have personally been involved in many, many discussions about the requirements for a SCADA graphical client, and to my knowledge, this meets every one of them and then some. Some of the features which I did not describe above include timed graph annotations, automatic alerting, exporting data to CSV files, embedding panels in other HTML files and more. Because it is open source, any feature that is not already included with Grafana will most likely exist as a plugin, and if not, could be easily written.

The team should strongly consider adopting Grafana as its default SCADA client for all data visualization and monitoring purposes. A server could be set up in the dyno room and provide live insight of tests being performed to any number of computers on the network. It is easy enough to use that each subsystem could have its own, dedicated dashboard, written by the team in charge of it, and displaying only the data needed for a given use case or test.