## 1 SETUP

### Hardware

The hardware required to run the GBB CAN Simulator is as follows:

* Raspberry Pi 3b+ or 4
* USB/CAN Adapter (The system has been tested with the adapters from CANUSB.com)

### Software

The Raspberry Pi should be configured with the latest version of the raspios. Development was done using the ‘2020-08-20-raspios-buster-armhf-full’ image.

Once the Raspberry Pi image has been installed and the user has logged into a bash shell, the can-utils package needs be installed. This can be done by

sudo apt-get install can-utils

These utilities are required by the GBB CAN Simulator

Once the source code has been added to the Raspberry Pi, a simple ‘make’ command will build the application and supporting libraries.

At this point the application is ready to be run.

Side note. I found sometimes git does not always handle file permissions correctly. All the slcan\_ files and restart need to be executable on the Linux system. Also, if you find you are problems with a script check that git did not add \r characters. If it did these need to be removed for the scripts to run correctly. If found ‘tr -d “\r” < filein > fileout’ does the trick.

### WWW Page Build

The HTML/Javascript code implementation started as an experiment. I grouped functionality into categories, WebSocket handling, user input handling, device management, etc. A directory was created for each category and any new function created for the category of functionality was add as a new file in the appropriate directory. No more than one function per file. Each directory contained a file that was the name of the directory with a .js suffix. This contained any publicly available data used by the category.

All the directories reside in a single directory under the base www directory call ‘Files’. There is a little utility call ‘bundle’ which collects all the individual files under ‘Files’ and its subdirectories and amalgamates them into a single file call “scripts.js”. This is the file that gets downloaded to the browser. The www directory contains an Makefile that build bundle and then runs to create “scripts.js”.

The JavaScript is straight JavaScript code, no jquery or similar frameworks. There is really nothing too complicated going on other some user interaction and a bit of communication with the Raspberry Pi.

At first glance this may seem cumbersome but I found, in practice it makes things easy to deal with. I know right where the code for a function is nice when using simple tools to edit the code and don’t have (or need) the functionality of an IDE to help keep track of things.

The www build is not invoked by the main application build although it should be. This will happen in a future release.

## 2 COMMAND LINE OPTIONS

*cansimws* is designed to be run from the command line. It has a handful of command line options.

**options**

**-b --baysfile** : Specify the bays information file (default Bays.json)

**-d --devicedef** : Specify the device definition file (default DeviceDefs.json)

**-h --help** : Display this information

**-H --httpport** : Specify HTTP listening port (default 8000)

**-mc --monitorcan** : Monitor CAN requests

-**mw --monitorweb** : Monitor web page requests

**-w --webdir** : Specify the base www directory (default www)

**-W --websocketport** : Specify Web Socket listening port (default 8001)

If the standard HTTP port of 80 is desired it may be set using the -H, however *cansimw* must be run as root.

If the default options are used including the default ‘www’ directory, it must be in the same directory from which the application is run.

Before the *cansimws* program can function, the CAN USB device must be plugged into the Raspberry Pi and the CAN interface started. Several shell scripts exist for start and stopping the interface.

slcan\_add will determine the USB port to which the CAN USB device is plugged into and create interface call slcan0 which is used by the *cansimws.* If slcan\_add is not run or no CAN USB device is plugged in, the application will start but display a warning message about the missing interface.

slcan\_remove will remove any interfaces added with sclan\_add.

slcan\_show will display the name of the interface added with sclan\_add if one was added.

A simple script ‘restart’ will perform a remove interface, add interface and start the application

## 3 USER INTERFACE

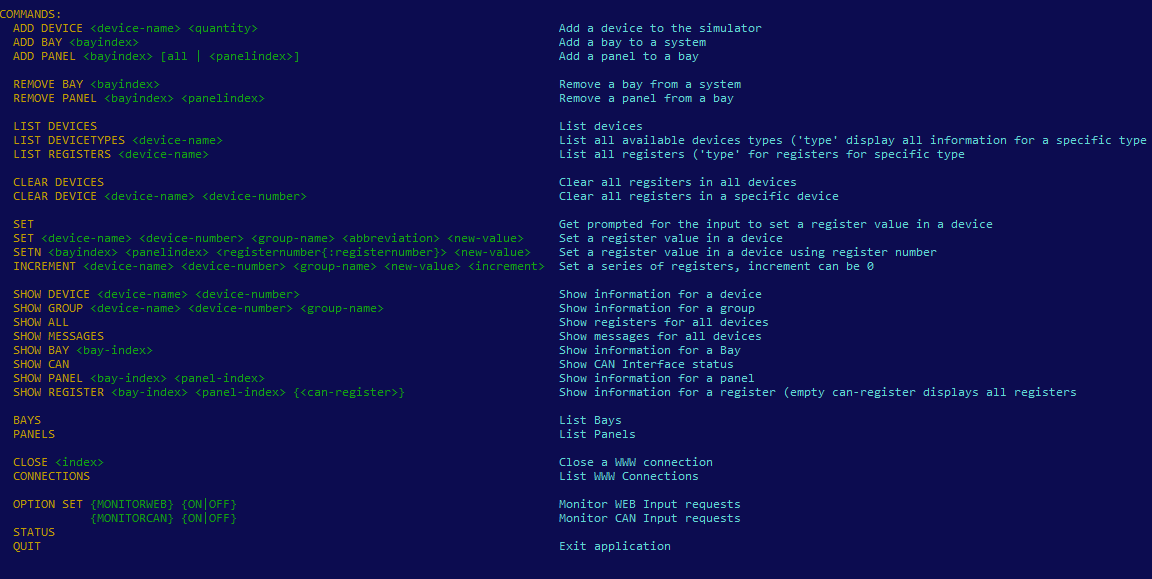


Figure 1

|  |  |  |
| --- | --- | --- |
| **ADD** | **DEVICE** | Add a new device by device name. An optional quantity can be supplied which causes a number of devices to be created. |
| **ADD** | **BAY** | Add bay with a bay index. |
| **ADD** | **PANEL** | Add a panel with a specific index to bay. |
| **REMOVE** | **BAY** | Remove a bay from the system. |
| **REMOVE** | **PANEL** | Remove a panel from the bay. |
| **LIST** | **DEVICES** | List instantiated devices. |
| **LIST** | **DEVICETYPES** | List the device types read in the Device Definitions file at start up. |
| **LIST** | **REGISTERS** | List all defined registers and message for a device |
| **CLEAR** | **DEVICES** | Clear the values in all registers in all instantiated devices. |
| **CLEAR** | **DEVICE** | Clear the values in specifically instantiated devices. |
| **SET** |  | Set the register value for a device. |
| **SETN** |  | Set the register value for a specific bay/panel. |
| **INCREMENT** |  | Increment a set of registers. |
| **SHOW** | **DEVICE** | List the registers associated with an instantiated device. |
| **SHOW** | **GROUP** | List the registers within a specific group associated with an instantiated device. |
| **SHOW** | **ALL** | List all registers for all instantiated devices. |
| **SHOW** | **MESSAGES** | Show all the message defined in the “messages” section in a device definition for all devices. |
| **SHOW** | **BAY** | Show the Bay and panel information for a specific device. |
| **SHOW** | **CAN** | Display the number of CAN message sent and transmitted. |
| **SHOW** | **PANEL** | Show the information a panel within a bay. |
| **SHOW** | **REGISTER** | Show a specific register for a panel within a bay., |
| **BAYS** |  | Display information for bays. |
| **PANELS** |  | Display information for panels. |
| **CLOSE** |  | For the closing of an active HTML connection. |
| **CONNECTIONS** |  | Display the active HTML connections. |
| **OPTION** | **SET** | The available system options. |
| **STATUS** |  | Display a minimal amount of system information. |
| **QUIT** |  | Exit the application. |

## 4 WEB USER INTERFACE

The application contains a simple HTML Interface. Two TCP ports are used in this application, one for the HTTP Server (Default 8000) and one for the Websocket communication (default 8001). Each of these ports may be changed using the command line interface. Using the default values, the unit is accessed via the URL:

<rpi-network-address>:8000

On a system that has not been configured the page shown in *Figure 2* is displayed. If a bay and panels has been configured, the bay and panels are displayed as shown in *Figure 8.*

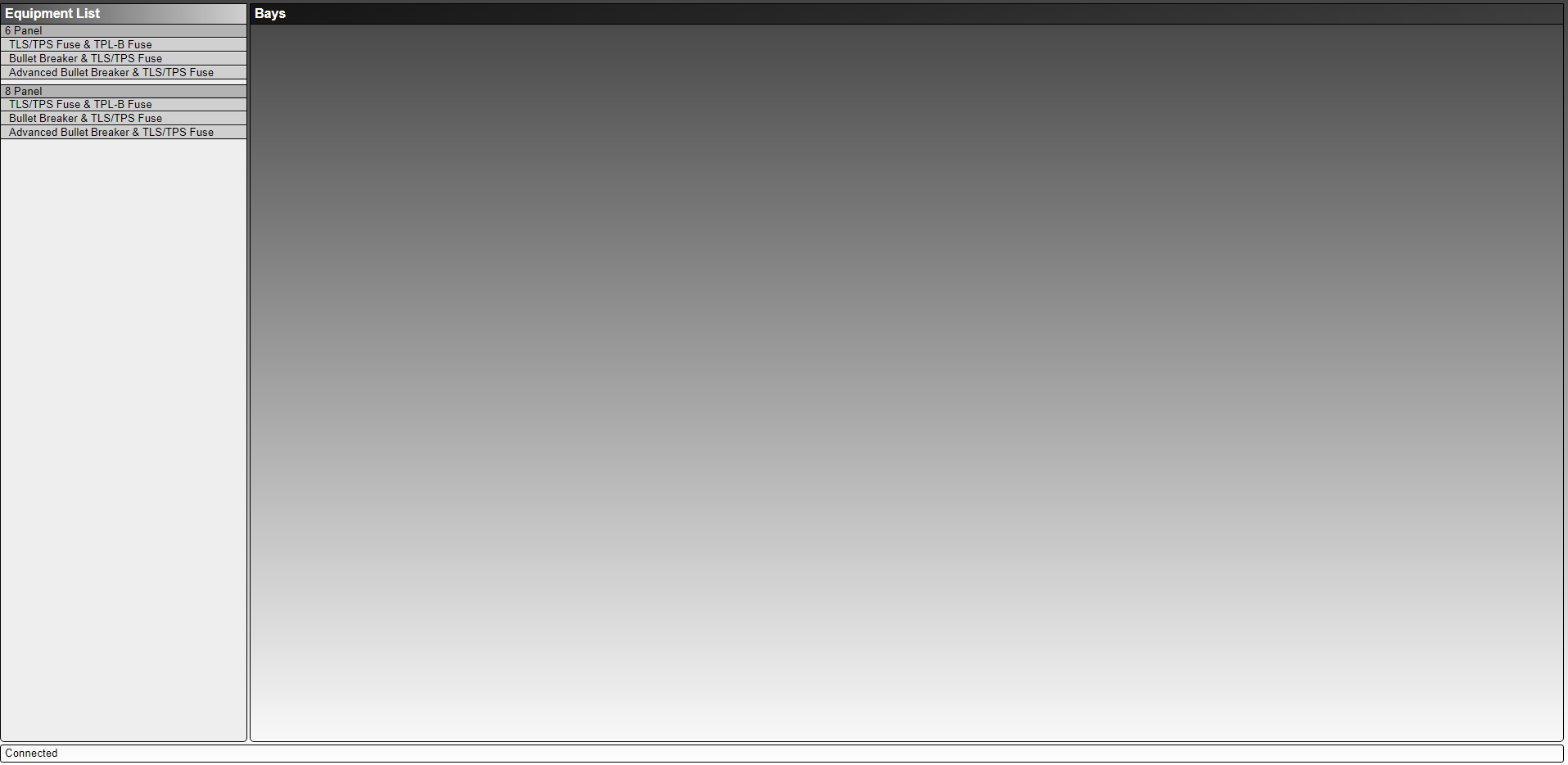


Figure 2

To add a bay to the system, click and drag one of the bay types (6 Panel or 8 Panel) to the **Bays** area as shown in *Figure 3.*

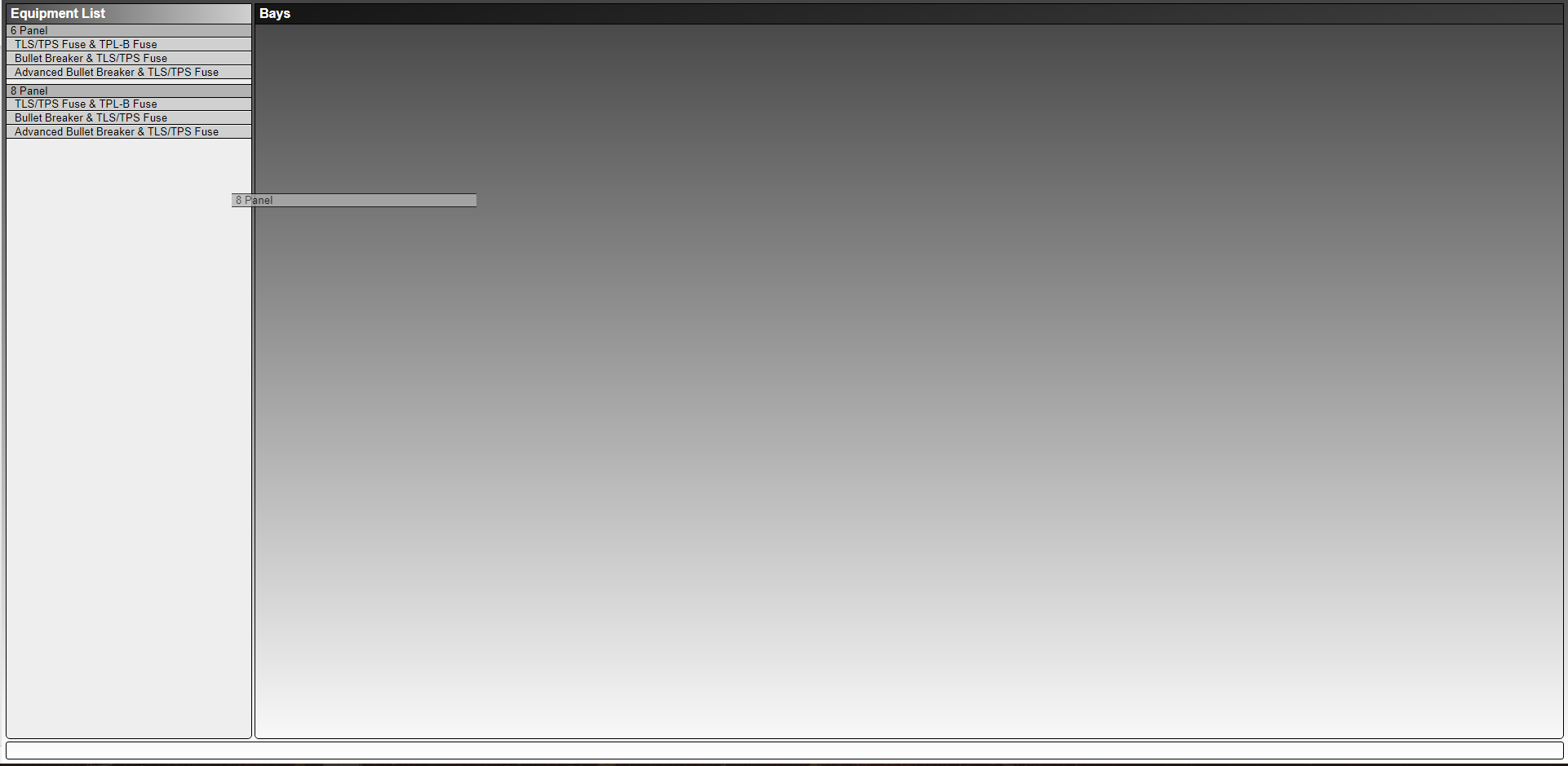


Figure 3

The application will add an unpopulated bay as in *Figure 4.*

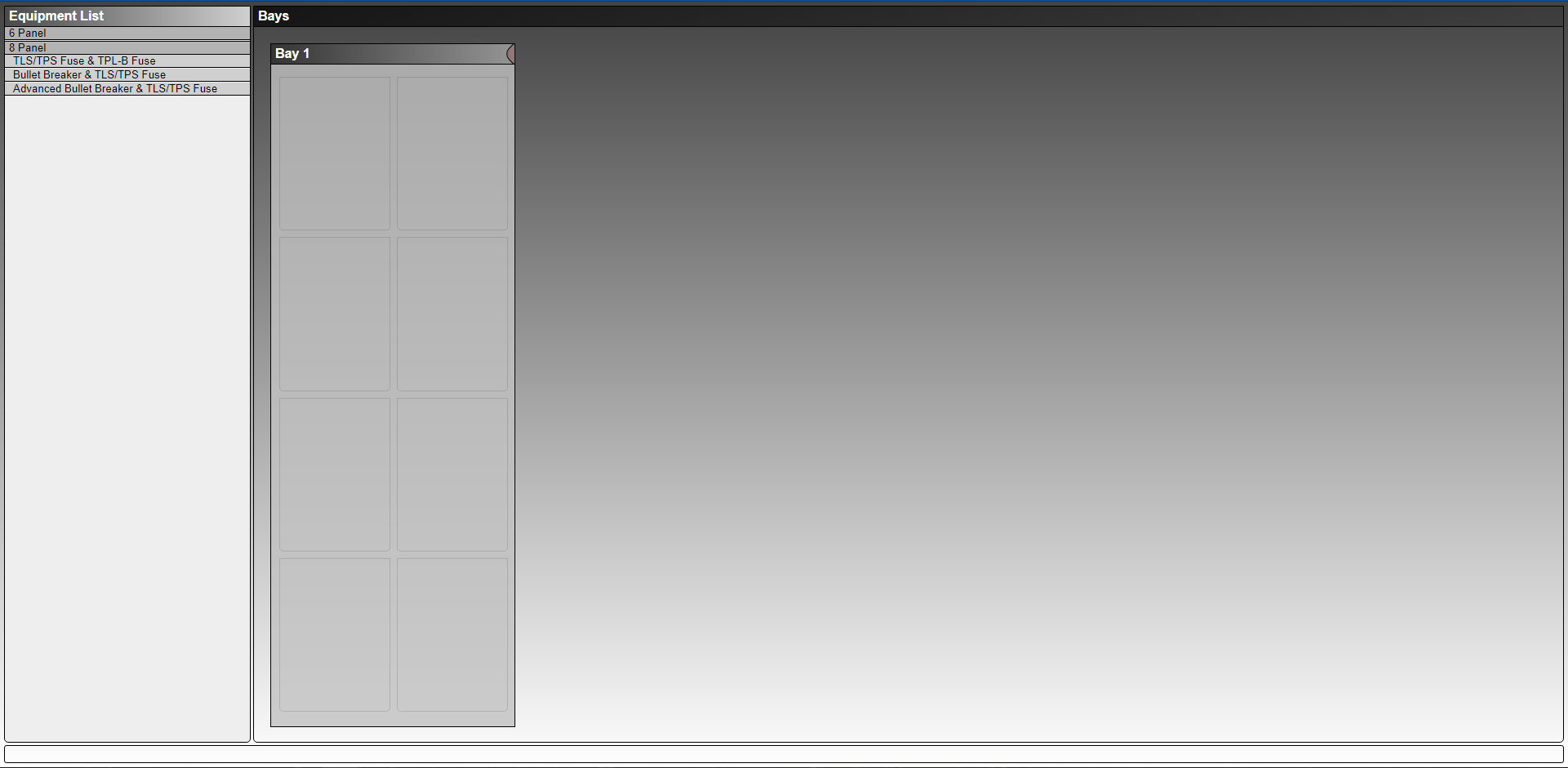


Figure 4

The application will create a simulate SMDUE whose register values may be changed via the command line UI or the HTML Page.

To change the registers here, click on the **Configure** button (the red semicircle on the top right corner of the simulated bay). This extends the bay display area and the categories of SMDUE registers are displayed along with the bay description information. The grouping of the register categories is determined in the Device Definitions file.

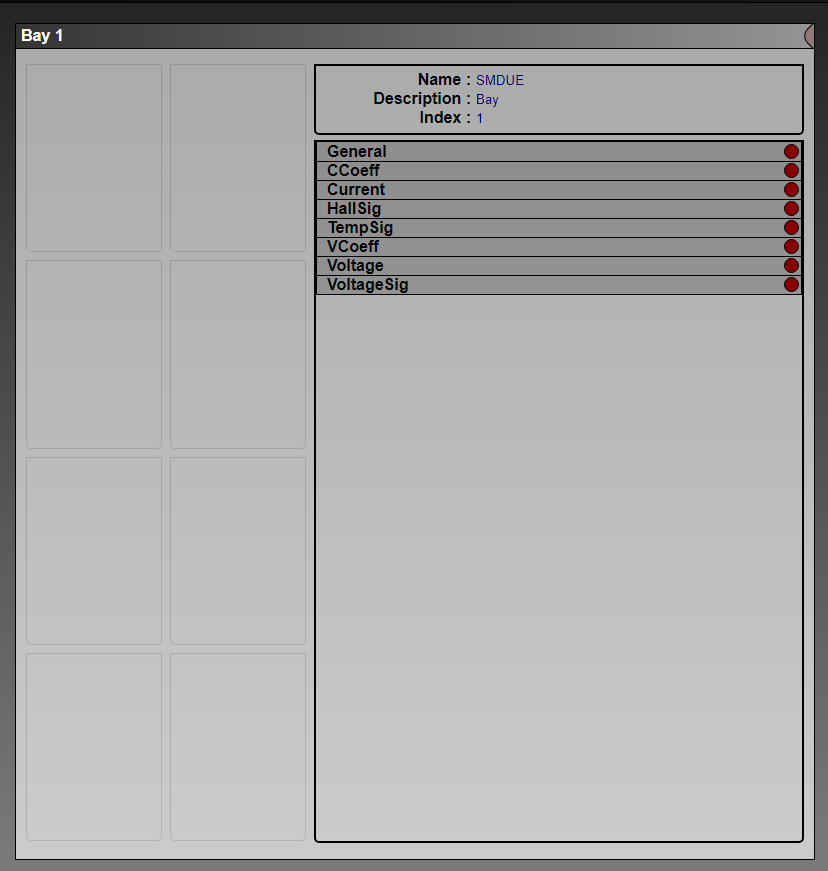


Figure 5

To change a specific register value, open the section to which the register belongs by clicking the red circle to the right of the section name. This drops down a list of the registers in the section along with input box to change the register values. The circle also changes color to green. To close the section, click on the green circle. *Figure 6* shows a bay with two sections open.

In the default Device Definitions file, the sections are created to logically group values together and to allow the sections shown here. The sections are used here to prevent screen clutter.

Register values are changed according to the data defined for the register in Device Definition file. The most commonly used value is a floating point value. Some values are ascii strings of up to 4 bytes or 32 bit hex values depending upon the register definition. See the Device Definitions section for a list of the available formats. The values are display in the appropriate format and when the user changes the values, the data is expected in the correct format. In the current application, no check is made to insure the data is in the correct format before being transmitted to the Raspberry Pi Server. In the current application the user is not informed that a data field is incorrect, or the data was rejected by the server. Plans are for the next round of changes to include this functionality.

To commit any changes made in the register input fields, click the red semicircle at the top right of the bay display area. This collapses the bay information page to its original view of just the bay panels. Currently, there is no way to cancel the commit operation if changes have been made to the register input fields. Once the register portion of the bay configuration is closed, any alter input field is seen as a change and applied.

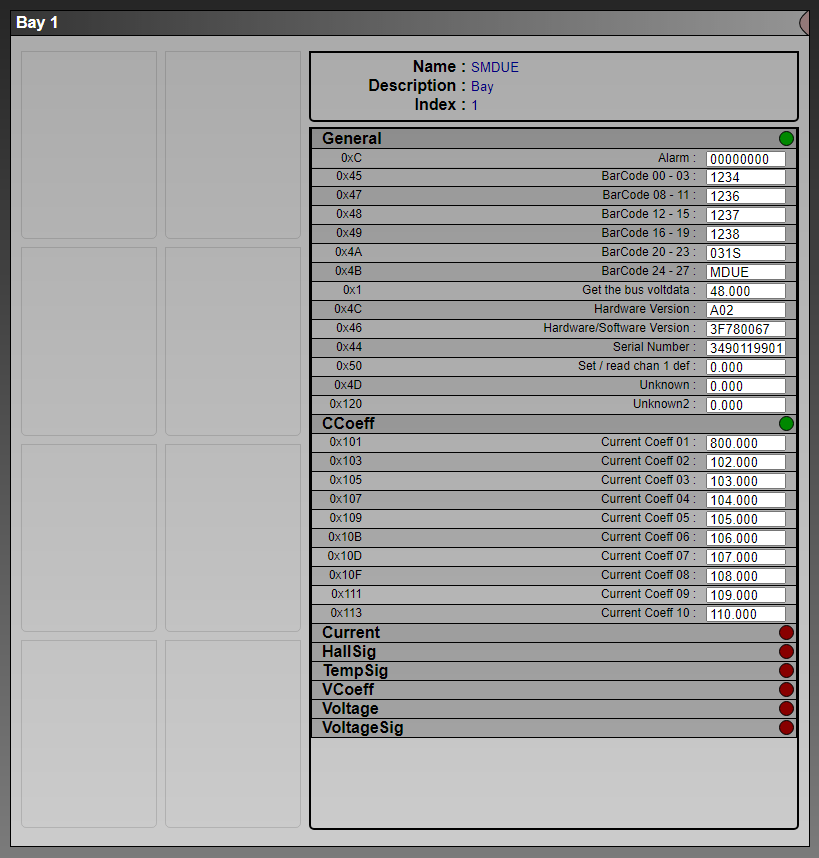


Figure 6

To add a panel to the bay, click on one of the available panel types from the Equipment List on the left and drag it to an empty panel location in the bay as shown in *Figure 7.*

Since only the advanced panels contain an SMDUH2 unit, the other two panel types are in effect just placeholders for panels in an actual panel and serve no current functional purpose in the simulator.

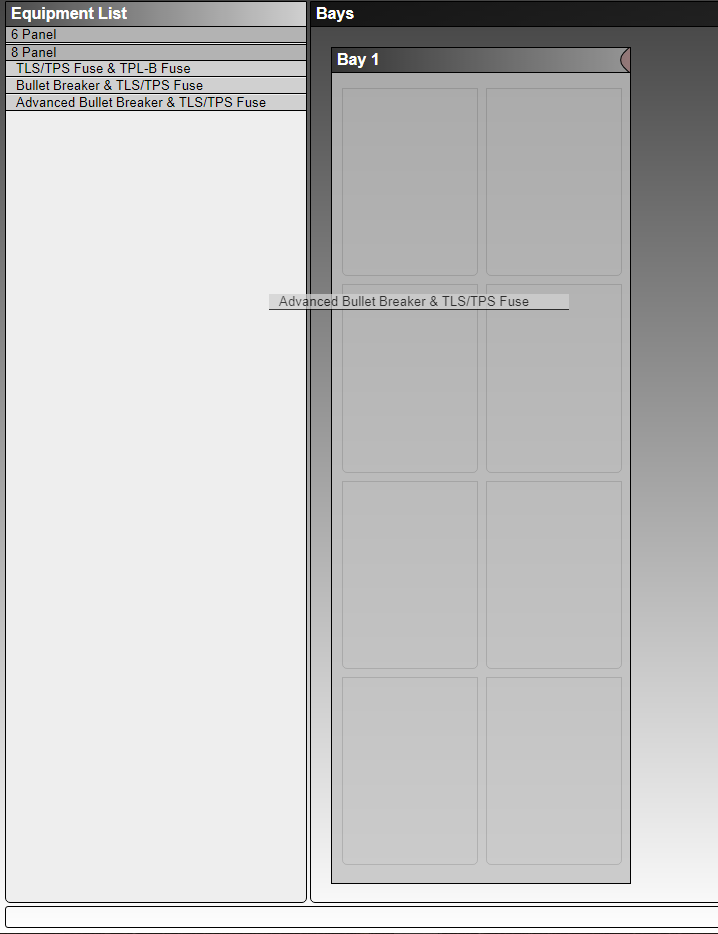


Figure 7

Once the panel type has been dragged to an empty panel location, the bay display will appear as in *Figure 8.* The bays are numbered 1 – 8 and laid out as follows :

**7 8**

**5 6**

**3 4**

**1 2**

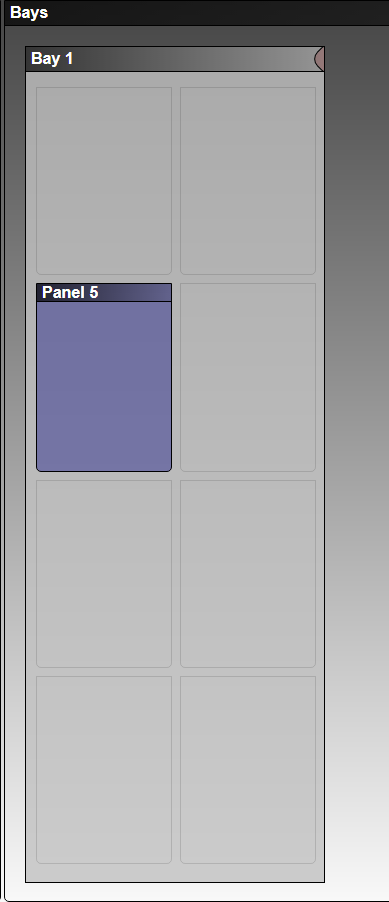


Figure 8

For advanced panels, to change the SMDUH2 registers, click on the panel image. A section like that of the bay/SMDUE is displayed as shown in *Figure 9.* As with the bay, unit identification information is displayed along with sections of registers. The registers for the panels are managed and edited in the same was as those for a bay. The same method of closing the extended panel display and the same restrictions on editing fields also apply.

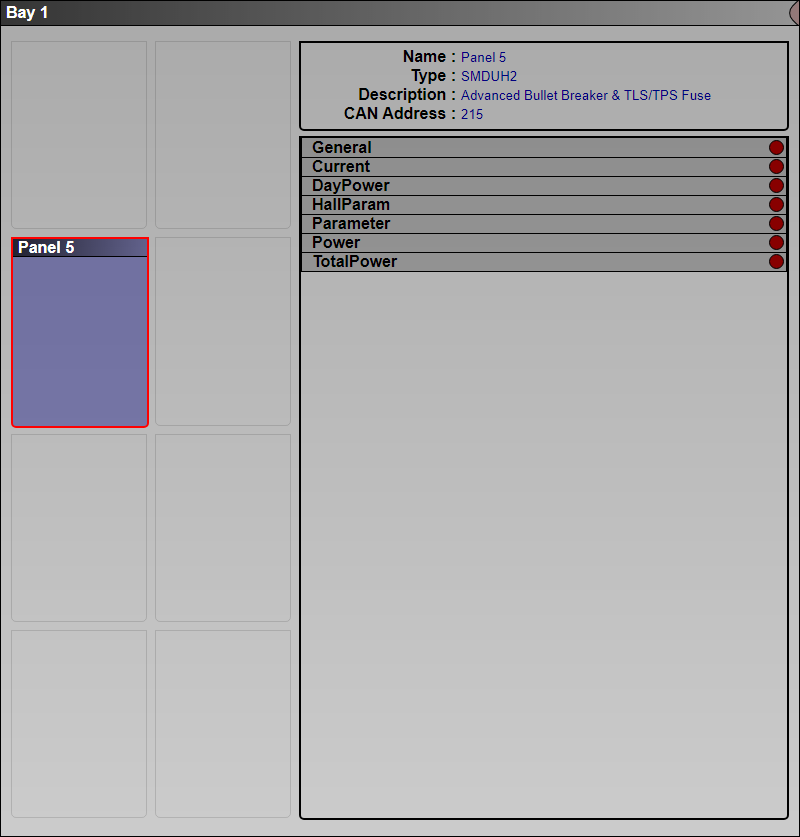


Figure 9

To remove a panel from a bay, click on and start to drag a panel as shown in *Figure 10*. As the panel begins to be dragged, a trash can will appear. Drag the panel to the trash can drop it by release the mouse button. This will remove the panel from the bay.

All changes made vie the HTML interface are sent to a server on the Raspberry Pi where the changes are applied to the internal data store and the changes are saved in a file on the Raspberry Pi’s SD card. It should be noted that changes made to a bay, panel or register made via the command line interface will not be transmitted to any HTML interface session running while the command line changes are being made. Provisions for doing this are in place and will be seen in a future version of the application.

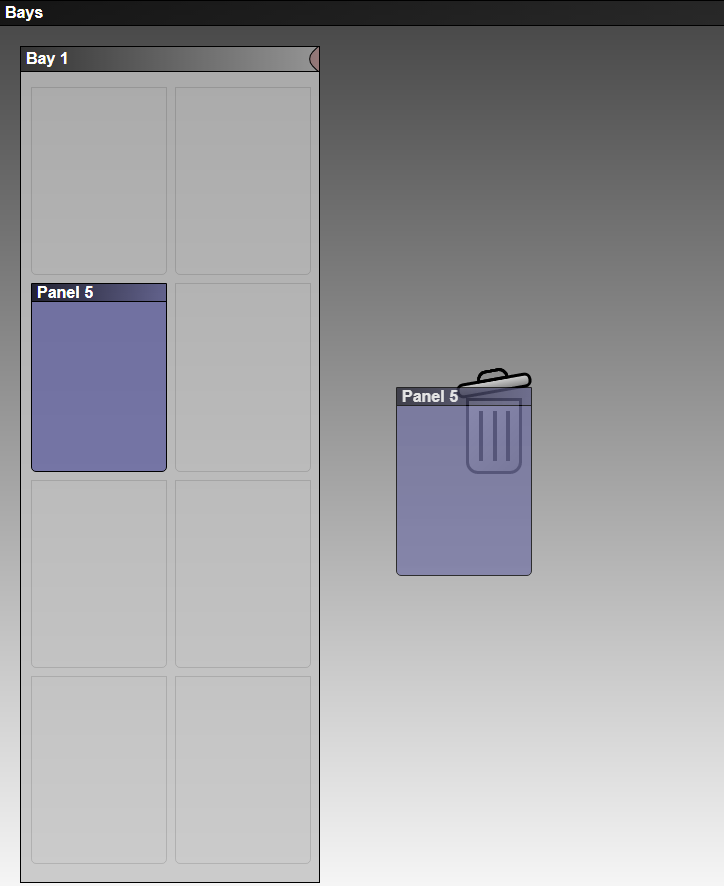


Figure 10

## 5 DEVICE DEFINITIONS

The application has no hardcoded information about any of the GBB units that communicate using the CAN protocol. Instead, a mechanism for defining the elements for a unit have been created. A CAN unit protocol can be defined and encoded in a JSON format which the application reads at start up. The HTML interface uses a Bay/Panel configuration to represent two of the CAN unit types, specifically an SMDUE and SMDUH2. While the Command Line interface also uses this model, it allows for the creation of any type of CAN unit defined in the JSON file. Values for this unit may be set at the command line and the registers associated with these units are visible to the NCU when communicating with the simulator.

### File Layout

The following shows a sample of definition

[

{

"name" : "SMDUE",

"protocol" : 166,

"description" : "Bay",

"startingcanaddress" : 219,

"regs" [

{

"valuetype" : 1,

"msgtype" : 65,

"abbrev" : "Bus voltage",

"groupsort" : "0General",

"group" : "General",

"name" : "Get the bus voltdata",

"initvalue" : "0x42400000",

"format" : 0,

"displaylabel" : "Get the bus voltdata"

}

]

"messages" : [

{

"request" : 0,

"name" : "Read real time data",

"responses" : [ "0x01", "0x02", "0x03", "0x04", "0x05", "0x06", "0x07", "0x08",

"0x09", "0x0a", "0x0b", "0x0c" ]

}

]

}

]

The fields are defined as follows:

|  |  |  |
| --- | --- | --- |
| **name** | The name of the unit (usually the externally known name SMDUH2, SMDUE…) | |
| **protocol** | The protocol number defined in the document specify the unit. | |
| **description** | A simple text description of the unit. | |
| **startingcanaddress** | The default starting CAN address used by units of this type. | |
| **regs** | An array of all registers used in this protocol. | |
|  | **valuetype** | The register number associated with this value. |
|  | **msgtype** | The message type used when responding to a request for this register. |
|  | **abbrev** | An abbreviated form the name. |
|  | **groupsort** | A modified version of the group used for sorting. This allows the groups to be sort by in an arbitrary order. |
|  | **group** | The group to which a register belongs. |
|  | **name** | The name of the register. |
|  | **initvalue** | The initial value assigned to the register when a new new unit is initialized. |
|  | **format** | The format of the value represented by this register  0 – Floating point number  1 – 3 byte ascii string  2 – Variable length string (< 4 bytes, used for serial numbers)  3 – A 4 byte ascii string  4 – A 4 byte hex value |
|  | **displaylabel** | The string used for a text label when displaying the registers value on www sites, text interfaces … |
| **message** | An array of message requests known by this protocol. | |
|  | **request** | The request value coming from the NCU. |
|  | **name** | The name of the request message . |
|  | **responses** | An array of register value types. These values types are messages defined in the ‘regs’ section. |