Contents

[Overview 3](#_Toc448854727)

[Introduction 3](#_Toc448854728)

[Future Integrations 4](#_Toc448854729)

[Physical Wiring 4](#_Toc448854730)

[Raspberry PI Add-on Hardware 5](#_Toc448854731)

[ECU Bridge Software 5](#_Toc448854732)

[Transcoding 7](#_Toc448854733)

[Usage 9](#_Toc448854734)

[Command Line Access 9](#_Toc448854735)

[echo 9](#_Toc448854736)

[status 9](#_Toc448854737)

[channels 10](#_Toc448854738)

[filter 11](#_Toc448854739)

[patch 12](#_Toc448854740)

[transform 13](#_Toc448854741)

[stop 14](#_Toc448854742)

[REST API 14](#_Toc448854743)

[[GET] /rest/ecubridge/Status 14](#_Toc448854744)

[[GET] /rest/ecubridge/Map 15](#_Toc448854745)

[[POST] /rest/ecubridge/SetFilter.json 16](#_Toc448854746)

[Direct Data Monitoring 16](#_Toc448854747)

[Web Access 17](#_Toc448854748)

[Status 17](#_Toc448854749)

[Channel Mapping 17](#_Toc448854750)

[Testing 18](#_Toc448854751)

[Graphing 19](#_Toc448854752)

[Connecting with a Phone 20](#_Toc448854753)

[Connecting with Your Desktop or Laptop 22](#_Toc448854754)

[System Setup 25](#_Toc448854755)

[OS Install 25](#_Toc448854756)

[Real Time Clock (RTC) 25](#_Toc448854757)

[FTDI RS-232 Quad Conversion Cable (USB) 27](#_Toc448854758)

[Apache / PHP 28](#_Toc448854759)

[WI-FI Adaptor (USB) 28](#_Toc448854760)

[ECU Bridge 33](#_Toc448854761)

[Installing ecubridge 33](#_Toc448854762)

[Installing ecudatalogger 35](#_Toc448854763)

[Installing rrdcached 35](#_Toc448854764)

# Overview

## Introduction

The ECU (Engine Control Unit) Bridge is a flexible means of bringing sensor data from legacy or non-standard equipment to modern data loggers and analysis tools. A Raspberry PI is used as a flexible and affordable way to physically wire the old equipment with the new. Software onboard the Raspberry PI (the ECU Bridge software) handles the details of mapping the legacy equipment inputs to the modern equipment outputs. Together the Raspberry PI and the ECU Bridge software are what we mean by “ECU Bridge”.

The ECU Bridge can even serve as a software defined ECU – potentially you can simulate data that would come from another engine, or you can bring together sensors to define your own ECU without actually having an ECU. The end result is that you can flexibly do data logging and analysis for equipment that normally would not be compatible at all with modern data loggers and analysis software.

In our project we are bringing in data from a Honda ECU via an analog/digital data logger (DL-32); from a “Chump Car” racing league car that isn’t compatible with modern data loggers. The data is then sent on to an AIM Solo DL and finally to an on dash SmartyCam camera.

While our project is specific to this equipment, the ECU Bridge is designed to be readily extended to other kinds of equipment or do other kinds of ECU translations.



Raspberry PI 2 (Model B)

AIM Solo DL

SmartyCam HD

Insert Your Chump Car

DL-32 A/D Logger

The RPI2 (w/Software) allows us to integrate legacy equipment with modern data loggers and performance analysis tools. Ultimately SmartyCam HD shows you RPM, Oil temperature etc. overlaid on your actual race video so you can easily analyze race performance.

## Future Integrations

Currently our focus is on getting sensor and ECU data from an old Honda car to the SoloDL and SmartyCam. However, the ECU Bridge could easily be extended further:

* In one of the remaining RS-232 ports, send data out to a radio link in addition to sending to the SoloDL port. So you could have a live feed from the car to the paddocks and be making real-time adjustments based on car performance.
* In one of the remaining RS-232 ports hook up other sensor modules. As long as the sensor or alternative ECU supports RS-232 in some fashion, you are good to go!

## Physical Wiring

The motor vehicle industry commonly uses a variation of RS-232 protocol/wiring called CAN or CAN Bus. Essentially its RS-232 but specialized for cars and other vehicles. Fortunately the equipment we are integrating doesn’t use full RS-232 capabilities; no modem control lines are used for example, its just raw RX/TX lines. So, as long as you can integrate with an RS-232 connector (basically a D-Sub 9-pin connector), and a UART…you are good to go.

In our set we use a USB to RS-232 conversion cable that allows us to work with 4 separate RS-232 connectors via a single USB plug. We currently use only 2 of them:

1. Input from the DL-32.
2. Output to the Solo DL.

The USB conversion cable we use is the FT4232H, which uses a UART chipsets that the Linux kernel has very good support for already. So connecting to the external devices requires very little effort; just make sure they are wired to the RX/TX lines of D-Sub 9 pin connectors, and plug them into the USB conversion cable. After that the ECU Bridge software takes over. You can send and receive on any of the RS-232 ports, but obviously the device must support either operation, and you must send or receive data with the appropriate protocol.



Fortunately both the DL-32 and SoloDL have documented specs for the protocol they use. Nether one is very complicated, but the SoloDL has some timing requirements; the various data channels must be sent with a particular frequency. Conversely the DL-32 has all its channels read at once.

NOTE: you will also need power ☺ A 5V DC adaptor ending with a micro-usb plugin for the RPI2, and for the DL-32, you’ll need a 12V DC power supply; something that is normally used for CB radios or the equivalent of a car batter. Such power supplies are easy to find at Amazon or your local electronics store. Personally I used a Pyramid PS3 3-Amp 12-Volt Power Supply with a chopped molex connector (that I wasn’t using from an old computer)

## Raspberry PI Add-on Hardware

A couple of add-ons are used to make life easier when working with the RPI2; neither takes away from system performance or available GPIO pins.

1. PI-Face Real Time Clock (RTC) – a batter backed hardware clock that allows us to have consistent time stamping onboard the RPI2. This is critical for the onboard data logging and any debugging that happens via the logs.
2. WI-PI Wi-Fi Adaptor – allows for wireless access to the RPI2. This in turn allows you to work with the RPI2 using a smart phone or tablet. If you are at the track you may not allays have a laptop handy!
3. FT4232H Quad RS-232 / USB cable. This cable allows us to not only connect to multiple RS-232 ports while using a single USB port, but also handles voltage conversion/regulating between RS-232 levels and TTL (5V DC) used by the Raspberry PI. Very handy!

## ECU Bridge Software

The high level view of the software running on the RPI2:

DL-32

Solo DL

ECU Bridge (Raspberry PI 2)

ecubridge

(daemon)

ecudatalogger

(daemon)

rrdtool

(daemon)

Command Port

TCP

UDP

RS-232

RS-232

USB Bus

The ecubridge daemon is the main controller. It “transcodes” DL-32 raw data to normal data, then transcodes again to format for the SoloDL. At the same time it logs the raw, normal and output data to the ecudatalogger (backed by RRD Tool), and listens for both user commands and USB Bus events. The main event loop is **select()** based and tracks to a 100ms work cycle (10hz) to align with the AIM Solo DL protocol. The DL-32 has a different data rate, but we just drop the occasional sample; 10hz resolution is more than enough to be reasonably accurate.

We must pay attention to the USB Bus because when we power on, the USB cable might not be connected yet, or it may be connected / disconnected while the RPI2 is up and running. We have to be ready to start and or pause as needed.

From above you can see that we have 3 (three) software daemons:

1. ecubridge – this is the main controller. In addition to passing data from DL-32 to the Solo DL, it will listen for user commands on a TCP port (which may come from the command line or the Web GUI), and will also listen to the USB Bus so it can react intelligently if the USB cable isn’t plugged in yet, or if the USB cable is re-connected.
2. ecudatalogger – listens to the ecubridge (via multi-cast UDP) to capture the raw, normal and output data of the ecubridge as its transcoded (live) without interrupting or slowing down the ECU Bridge. Once we have the data, we pass on to RRD for actual storage and to later generate graphs etc. Note though we are currently configured to only capture the most recent 24 hours worth of data, so after a race, be sure to power off the RPI2 so that it doesn’t keep recording and wipe out the race data.
3. RRD Tool – this is the standard Open Source tool for data logging. Even though it doesn’t support millisecond level time resolution (it only goes down to 1second), we just average our recorded data (at 10hz) to 1hz and log that. The purpose of the on board data logging is for analyzing and debugging the transcoding of data not the actual performance of the race car, so a resolution of 1sec is acceptable. Other onboard data loggers were considered, but basically RRD is the best in breed and easiest / fastest to get up and running with.

These are all started automatically when the RPI2 is powered on. They each have their own configuration and logging (see the System Setup chapter). You can fully configure them as needed, and debug them using their log files.

## Transcoding

To allow the ECU Bridge to be easily extended, flexibly adapt to different kinds of inputs or work with an output other than the Solo DL, or potentially act as a software defined ECU, we’ve introduced the idea of a virtual channel:

Virtual Channel (1 .. 15)

DL-32

Solo DL

Transform

Transform

Filter

Filter

Source

Dest

Input Transform: convert voltage level to scaled value (i.e. an engine RPM value)

Raw Data

Output Data

Normal Data

Input Filter – manual override other filtering of input values.

Input / output patching, maybe channel 1 input should be wired to channel 2 output, etc.

Output Filter – allow for manual override.

Output Transform - align with AIM protocol

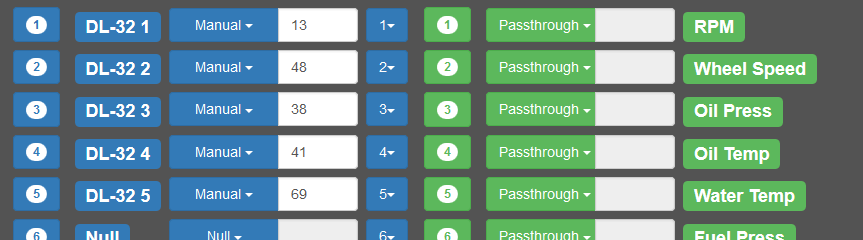
Once input data has been transformed and filtered, it is now in “normal” format. For example if you expect to see an RPM value of 1200, the same as what the Honda sees on its dashboard…this is where you would see 1200. At the input the value may be encoded as something else (i.e. a 12 bit voltage level), and at output it can be encoded as something else to match the AIM protocol. Its only in the middle where we expect data to be “normal”.

There is a virtual channel for each data channel the output device supports. In our project it’s the Solo DL, so there are 15 channels, ranging in kind from RPM, to water temperature to error flag. Other devices could potentially have different kinds of channels.

Virtual channels also give us a lot of flexibility, by defining in software what transformations happen on the input to the channel and the output of the channel, we can easily adapt to connecting different kinds of devices, or having different ways of scaling / interpreting the inputs and outputs. For example if the DL-32 was replaced with some other means of monitoring the Honda ECU in an analog way…we might have to change how we scale the input values to get the “normal” values we expect from the gauges on the car dashboard.

Also, at run time we may decide that we need to change which inputs go to which outputs…but we may not want to have to physically rewire the input setup. To allow for this kind of hot wiring, we support the idea of source/destination patching. In the middle of the virtual channel, we allow the normal data to be redirected from one virtual channel to another before it goes to the output side (filtering and transformation), and on to the Solo DL.

Example Channel Map



In this example all 5 DL-32 channels have been given manual overrides. For example The RPM channel has been pegged at 13. Notice that for Source and Dest, we have 1:1, that is no re-ordering of inputs to outputs. But the pull down for Source could be used to swap (for example) channel 1 input with any other channel input. So you can “hot wire” what DL-32 input is used for the RPM channel in the Solo DL.

Both input and outputs have transformers. For example   
“DL-32 1” would know how to interpret a 12bit voltage level value to be an appropriate scaled value for the range of RPM values for the Honda. Similarly on the output side “RPM” is a transformer that knows how to align with AIM protocol so that when you have a value of RPM=1200, the Solo DL will actually show 1200. The AIM protocol actually does various transforms for the different channels, presumably to keep them with the range of data types used in the Solo DL.

Simply by “patching” channels we can hot wire where input data goes. No need to get out the ‘ol tool box ☺

# Usage

## Command Line Access

Before running commands on the Raspberry PI 2 (RPI2), you’ll need to be connected. Please review the sections below on connecting with your phone, laptop or desktop if you aren’t sure how to connect to the RPI2.

To chat with the ECU Bridge via the command line you send ASCII commands (one line at a time) to the ECU Bridge command port (we’ve reserved TCP port 5999 for this in /etc/services). Because its just a TCP port, you can connect with telnet, netcat, socat, whatever your favorite TCP tool is. For the purpose of examples, we will use netcat.

Please keep in mind that because the ECU Bridge is in 100ms loop (it has to send data to the SoloDL every 100ms), you should not do long commands, otherwise some data points could be missed. The commands currently available are all very fast, so this shouldn’t be an issue, and normally you would not be issuing commands while driving the car anyways ☺

The general format for all commands is a comma (,) separated list of arguments, with the first argument being the command name and the second argument being the sub-command or parameters. No whitespaces are used/allowed, except for the newline (\n) to end the command.

### echo

Simply echo back whatever you put in:

chumpcar@chumpcar:~$ echo "echo,1,2,3" | netcat localhost 5999

123

END

chumpcar@chumpcar:~$

### status

Display a quick summary of how the ECU Bridge is doing and key statistics:

chumpcar@chumpcar:~$ echo "status" | netcat localhost 5999

status: OK

reads: 0

writes: 222908

uptime: 22190

cmds: 1

config: /etc/ecubridge/ecubridge.ini

log: /var/log/ecubridge

END

chumpcar@chumpcar:~$

NOTE: if you see “reads: 0” it means the DL-32 is not powered or not connected. If you see a status of DISCONNETED, it means the USB cable is unplugged.

### channels

This command and its sub-commands is very useful to see what the current channel configuration is for all 15 channels at once. To see the summary of channel configurations use the map sub-command:

chumpcar@chumpcar:~$ echo "channels,map" | netcat localhost 5999

DL-32 > [ DL-32 1][ Passthrough] .. [ Passthrough][ RPM] > SoloDL

DL-32 > [ DL-32 2][ Passthrough] .. [ Passthrough][ Wheel Speed] > SoloDL

DL-32 > [ DL-32 3][ Passthrough] .. [ Passthrough][ Oil Press] > SoloDL

DL-32 > [ DL-32 4][ Passthrough] .. [ Passthrough][ Oil Temp] > SoloDL

DL-32 > [ DL-32 5][ Passthrough] .. [ Passthrough][ Water Temp] > SoloDL

DL-32 > [ Null][ Null] .. [ Passthrough][ Fuel Press] > SoloDL

DL-32 > [ Null][ Null] .. [ Passthrough][ Batt Volt] > SoloDL

DL-32 > [ Null][ Null] .. [ Passthrough][ Throt Ang] > SoloDL

DL-32 > [ Null][ Null] .. [ Passthrough][ Manif Press] > SoloDL

DL-32 > [ Null][ Null] .. [ Passthrough][ Air Charge Temp] > SoloDL

DL-32 > [ Null][ Null] .. [ Passthrough][ Exh Temp] > SoloDL

DL-32 > [ Null][ Null] .. [ Passthrough][ Lambda] > SoloDL

DL-32 > [ Null][ Null] .. [ Passthrough][ Fuel Temp] > SoloDL

DL-32 > [ Null][ Null] .. [ Passthrough][ Gear] > SoloDL

DL-32 > [ Null][ Null] .. [ Passthrough][ Error Flag] > SoloDL

END

chumpcar@chumpcar:~$

### filter

This command allows you to do a manual override, you can peg an input or output filter at a certain value. After that the ECU Bridge will continue to send that data for that channel to the Solo DL and log it via the ECU Data Logger. You can set the input filter, which impacts the “normal” data and will then be filtered/transformed on output to go to the Solo DL. You can set the output filter, which is more direct to the Solo DL, it will not be transformed further, except by the SoloDL transform for that channel. All Solo DL output data is inverse transformed to make the transform used by the Solo DL protocol “cancel out”. If the ECU bridge then tries to send “42” the SoloDL will see “42”.

In this example we set the output filter for channel 3 to 12, which means the Solo DL will see “12” as the oil pressure.

chumpcar@chumpcar:~$ echo "filter,output,3,manual,12" | netcat localhost 5999

OK. Filter set.

END

chumpcar@chumpcar:~$ echo "channels,map" | netcat localhost 5999

DL-32 > [ DL-32 1][ Passthrough] .. [ Passthrough][ RPM] > SoloDL

DL-32 > [ DL-32 2][ Passthrough] .. [ Passthrough][ Wheel Speed] > SoloDL

**DL-32 > [ DL-32 3][ Passthrough] .. [ Manual (12)][ Oil Press] > SoloDL**

DL-32 > [ DL-32 4][ Passthrough] .. [ Passthrough][ Oil Temp] > SoloDL

DL-32 > [ DL-32 5][ Passthrough] .. [ Passthrough][ Water Temp] > SoloDL

DL-32 > [ Null][ Null] .. [ Passthrough][ Fuel Press] > SoloDL

DL-32 > [ Null][ Null] .. [ Passthrough][ Batt Volt] > SoloDL

DL-32 > [ Null][ Null] .. [ Passthrough][ Throt Ang] > SoloDL

DL-32 > [ Null][ Null] .. [ Passthrough][ Manif Press] > SoloDL

DL-32 > [ Null][ Null] .. [ Passthrough][ Air Charge Temp] > SoloDL

DL-32 > [ Null][ Null] .. [ Passthrough][ Exh Temp] > SoloDL

DL-32 > [ Null][ Null] .. [ Passthrough][ Lambda] > SoloDL

DL-32 > [ Null][ Null] .. [ Passthrough][ Fuel Temp] > SoloDL

DL-32 > [ Null][ Null] .. [ Passthrough][ Gear] > SoloDL

DL-32 > [ Null][ Null] .. [ Passthrough][ Error Flag] > SoloDL

END

chumpcar@chumpcar:~$

In addition to being able to set a manual filter, you can use “passthrough” and “null” (which don’t take any further parameters).

### patch

Use this command change the ordering of how the left side inputs map to the right side inputs. This allows you to hot wire which inputs go to which outputs. You can also set the ordering at startup via the ecubridge.ini file (which is already implemented). If you mess up, you can use the reset and default sub-commands respectively to reset the patch ordering to what it was in the .ini file or the factory default (i.e. 1:1, 2:2, etc.)

chumpcar@chumpcar:~/dev/chumpcar$ echo "**patch,swap,2,1**" | netcat localhost 5999

OK. swapped.

END

chumpcar@chumpcar:~/dev/chumpcar$ echo "channels,map" | netcat localhost 5999

DL-32 > [ DL-32 2][ Manual (42)] 2.. 1 [ Passthrough][ RPM] > SoloDL

DL-32 > [ DL-32 1][ Manual (3)] 1.. 2 [ Passthrough][ Wheel Speed] > SoloDL

DL-32 > [ DL-32 3][ Passthrough] 3.. 3 [ Passthrough][ Oil Press] > SoloDL

DL-32 > [ DL-32 4][ Passthrough] 4.. 4 [ Passthrough][ Oil Temp] > SoloDL

DL-32 > [ DL-32 5][ Passthrough] 5.. 5 [ Passthrough][ Water Temp] > SoloDL

DL-32 > [ Null][ Null] 6.. 6 [ Passthrough][ Fuel Press] > SoloDL

DL-32 > [ Null][ Null] 7.. 7 [ Passthrough][ Batt Volt] > SoloDL

DL-32 > [ Null][ Null] 8.. 8 [ Passthrough][ Throt Ang] > SoloDL

DL-32 > [ Null][ Null] 9.. 9 [ Passthrough][ Manif Press] > SoloDL

DL-32 > [ Null][ Null] 10..10 [ Passthrough][ Air Charge Temp] > SoloDL

DL-32 > [ Null][ Null] 11..11 [ Passthrough][ Exh Temp] > SoloDL

DL-32 > [ Null][ Null] 12..12 [ Passthrough][ Lambda] > SoloDL

DL-32 > [ Null][ Null] 13..13 [ Passthrough][ Fuel Temp] > SoloDL

DL-32 > [ Null][ Null] 14..14 [ Passthrough][ Gear] > SoloDL

DL-32 > [ Null][ Null] 15..15 [ Passthrough][ Error Flag] > SoloDL

END

chumpcar@chumpcar:~/dev/chumpcar$

In this example channel 1 had been manually pegged at 3, and channel 2 had been manually pegged at 42. But after the swap, you can see that DL-32 input side for the first and second channels are swapped…so now if you look in the “Online” dialog in Race Studio (the live feed from the Solo DL), you’ll see that RPM shows 42 and Wheel Speed shows 3 (since the two channels were swapped!)

You can swap any of the channels and swap as often as you like. However you are limited to swapping exactly one channel with exactly one other channel. Also keep in mind that the output channel side (the right side) must stay consistent, because that is tied to the AIM protocol. So when you swap channels, you are really swapping inputs on the left – the right side stays the same always.

If you mess up, you can reset to what the patch ordering was in the .ini file:

chumpcar@chumpcar:~/dev/chumpcar$ echo "patch,reset" | netcat localhost 5999

OK. patch table reset.

END

chumpcar@chumpcar:~/dev/chumpcar$

### transform

If you just want to quickly test what Solo DL is setting you can use the “transform” sub-command. Given the current setup:

chumpcar@chumpcar:~/dev/chumpcar$ echo "channels,map" | netcat localhost 5999

DL-32 > [ DL-32 2][ Manual (42)] 2.. 1 [ Passthrough][ RPM] > SoloDL

DL-32 > [ DL-32 1][ Manual (3)] 1.. 2 [ Passthrough][ Wheel Speed] > SoloDL

DL-32 > [ DL-32 3][ Passthrough] 3.. 3 [ Passthrough][ Oil Press] > SoloDL

…

You can quickly check what The Wheel Speed should be with:

chumpcar@chumpcar:~/dev/chumpcar$ echo "channels,transform,2,66" | netcat localhost 5999

3

END

chumpcar@chumpcar:~/dev/chumpcar$

Our input value of 66 replaced with 3, because the DL-32 input 1 is currently patched to channel 2, and that input has a manual filter of 3…so the wheel speed will be 3. You can also confirm by looking at the “Online” dialog in Race Studio (the live feed from the Solo DL).

### stop

You should probably never use this command. If you send this command the ECU Bridge daemon will do a controlled exit.

## REST API

To allow the web interface and other applications to easily integrate with the ECU Bridge, a REST API is provided. This style of API is easy to use, highly portable and can be easily integrated into any language that supports web programming…which is pretty much any modern language, scripted or otherwise.

The following REST endpoints are available and can be used to integrate with the ECU Bridge daemon.

### [GET] /rest/ecubridge/Status

Fetch the general status of the ECU Bridge. Does not take any parameters.

Example output:

{

"data": {

"status":"OK",

"reads":"0",

"writes":"607624",

"uptime":"60485",

"uphours":"16 hours 48 min 05 sec",

"cmds":"10",

"config":"\/etc\/ecubridge\/ecubridge.ini",

"log":"\/var\/log\/ecubridge"

},

"status":"OK",

"error":""}

### [GET] /rest/ecubridge/Map

Fetch a summary of how the channels are configured. Does not take any parameters.

Example output:

{

"data":

[{

"channel":1,

"inputtransform":

"DL-32 2",

"inputfilter":"Manual (42)",

"src":"2",

"dst":" 1",

"outputfilter":"Passthrough",

"outputtransform":"RPM"

},

...

{

"channel":15,

"inputtransform":"Null",

"inputfilter":"Null",

"src":"15",

"dst":"15",

"outputfilter":"Passthrough",

"outputtransform":"Error Flag"

}

],

"status":"OK",

"error":""

}

### [POST] /rest/ecubridge/SetFilter.json

Tell the ECU to set an input or output filter. Both filters are applied to data, but the input filter happens first. Note also that data is transformed from DL-32 inputs before it reaches the input filter and the Solo DL outputs are transformed (per AIM protocol) after the output filter.

Using the filters allows you to manually peg the value for a channel.

Required inputs:

* channel - must be channel to modify
* side - must be input or output
* kind - must be passthrough, null or manual
* value - (optional) the number to use

## Direct Data Monitoring

If you would like to watch the data coming off one of the ECU Bridge data (raw, normal or output) without having to wait for data to flow through the ECU Data Logger to the RRD files, you can stop (or not start at all) the ECU Data Logger service (see section below on Installing the ecudatalogger), and then just watch the data taps directly. Because they are multi-cast UDP ports, you’ll need some trickery to monitor, you can’t just do a netcat on an plain old TCP port.

For this kind of monitoring socat is likely the easiest method:

socat STDIO UDP4-RECV:6100,ip-add-membership=226.1.1.1:lo

You will then get a stream of data - in this case the raw data from the DL-32 (6100 is the raw data port) and it will be updated every 100ms (10hz), because the ECU Bridge sends out the current/most recent values of raw, normal and output to their respective data ports every 100ms…that is every time the ECU Bridge writes to the Solo DL.

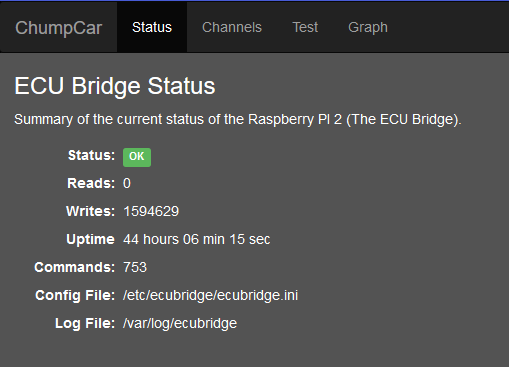
## Web Access

To access the web server (the GUI for the ECU Bridge), just point your favorite browser to the RPI2 IP address (see connecting with phone or computer below). The default website is setup to be the ECU Bridge web application. At the moment, it’s a simple interface intended only to provide basic control and status of the ECU Bridge. If the Wi-FI IP address isn’t working, make sure an Ethernet cable is hooked up and use the LAN interface; it’s the primary interface on the RPI2 and will be working even if the Wi-FI interface isn’t. If neither IP address is working…hook up an HDMI monitor and USB keyboard and look for problems in /var/log/syslog and dmesg.

The web GUI has 4 different pages, the home page is the Status page, but using the navigation menu at the top, you can access the channel map, real-time/live testing and graphing of recent data.

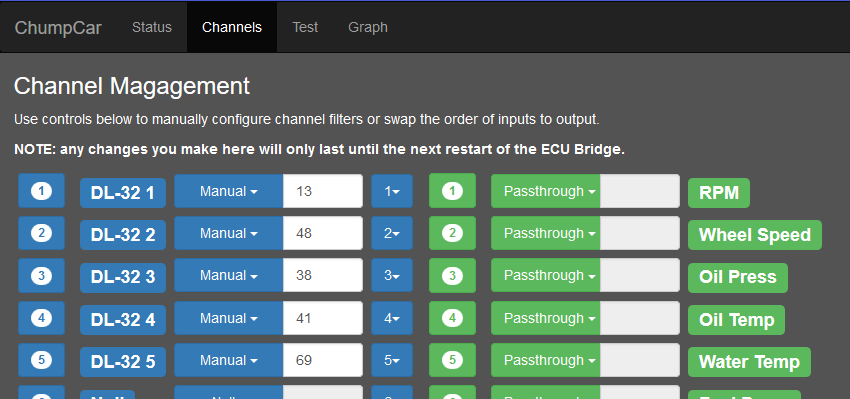
### Status

The status page allows you see a quick summary of the ECU Bridge health. If the USB Cable is unplugged, it will clearly indicate that.



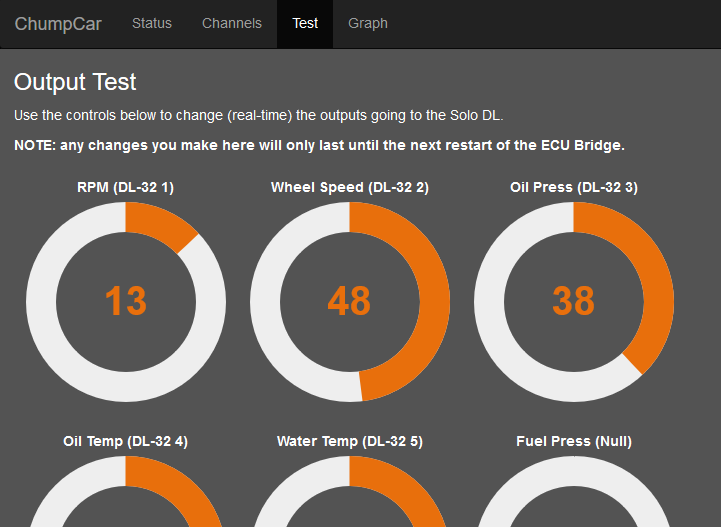
### Channel Mapping

From here you can manually set input or output filters and “hot wire” which inputs go to which outputs. Setting an input filter to a specific value, is pegging it for that output. This is handy for testing so you can confirm that that a given value is filtered and transformed correctly and then shown correctly in the Solo DL and SmartyCam.



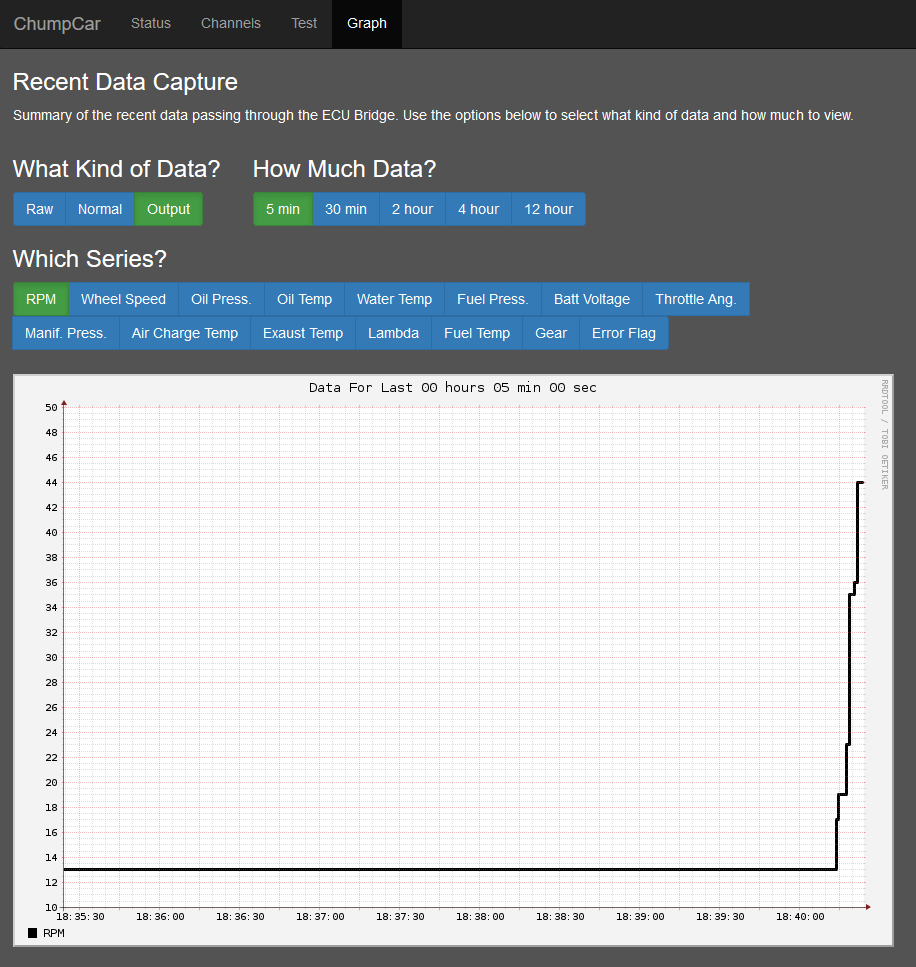
### Testing

This page has live dials or “knobs” for each Solo DL data channel. By using the knobs here you are interactively setting the input filter to manual (for that knob value) on the given channel. So if you play with the RPM knob, you should be able to see the RPM value change in the Solo DL and SmartyCam in real-time as you turn the knob.



### Graphing

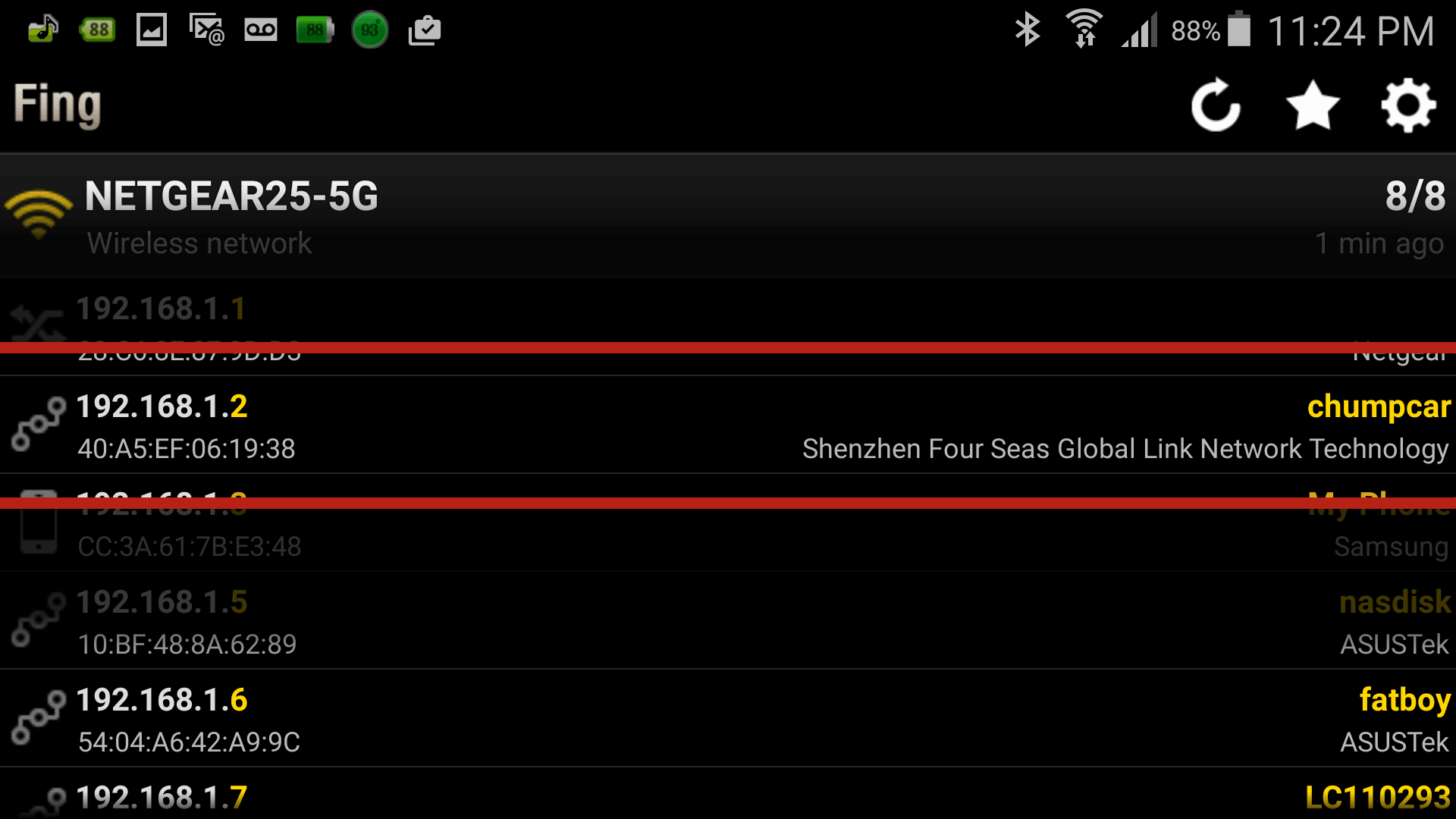
If you want to quickly see what the recent data has been in the ECU Bridge, you can use this page to easily select some data channels and how much of recent logged data to show. This is mainly to help you debug how transcoding is done in the ECU Bridge. Its resolution is limited to 1sec (because RRD can’t do sub-second resolution), so its not really precise enough for car engine performance analysis. But for debugging the ECU Bridge, it should be more than enough.



## Connecting with a Phone

To determine the IP address of the Raspberry from your phone (connecting wirelessly), you can use Fing:

<https://play.google.com/store/apps/details?id=com.overlook.android.fing&hl=en>



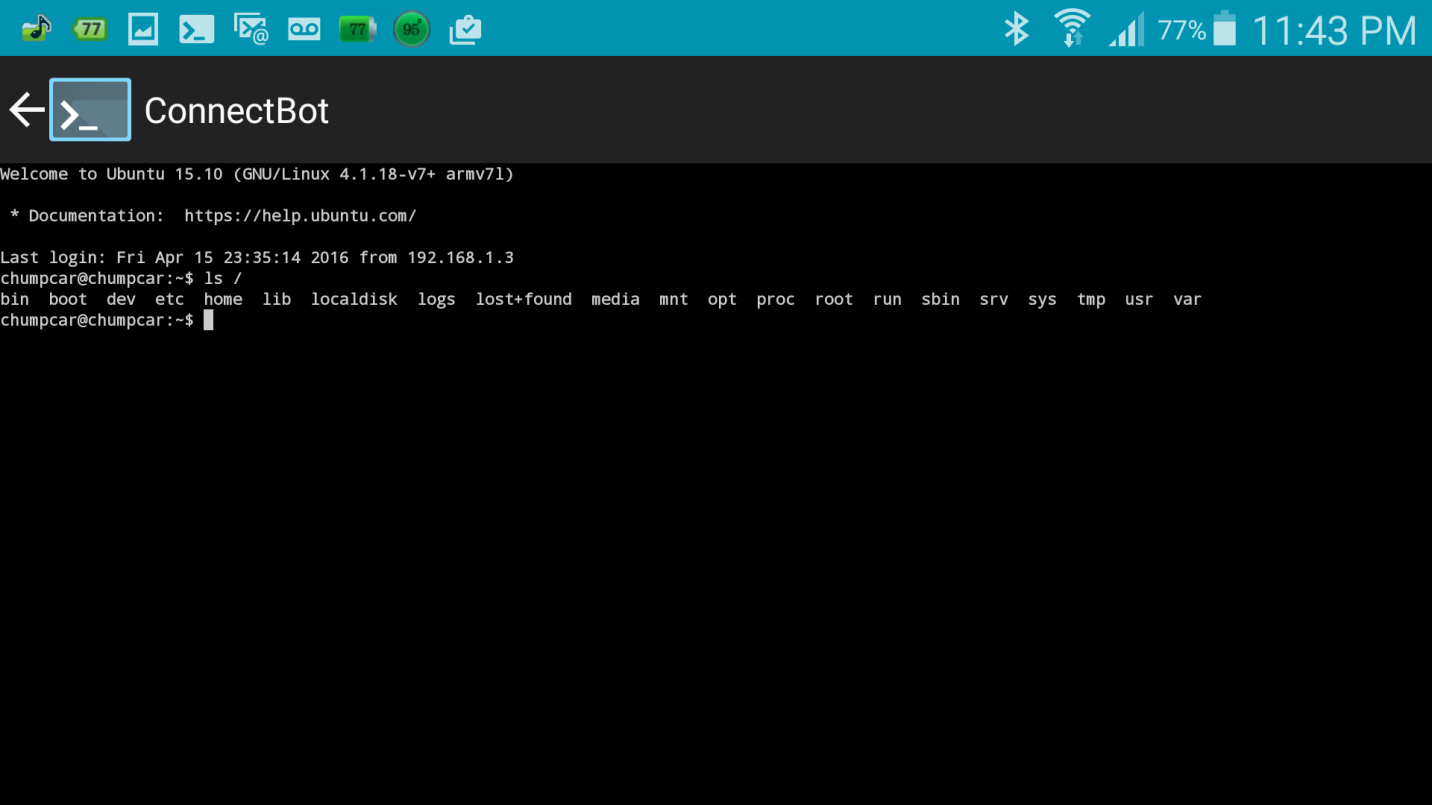
Other apps for IP scanning from your phone are also available.

Once you know the IP to connect to, you just need an app for your phone that understands ssh. Obviously you can also use your web browser to view the web site on the RPI2, regardless of the browser being on your phone, tablet, desktop computer or laptop.

For SSH on your phone, ConnectBot can be used:

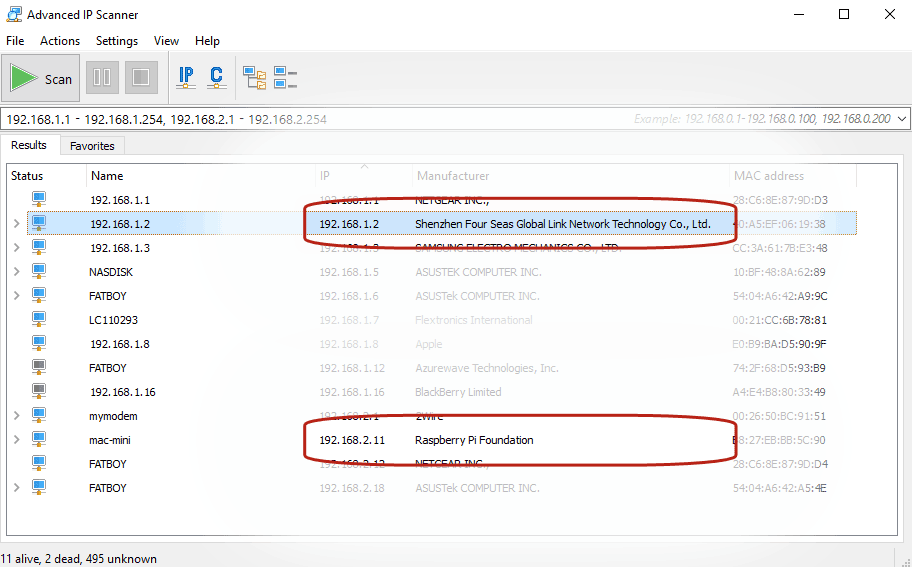
https://play.google.com/store/apps/details?id=org.connectbot&hl=en

Its actually handy for more than just the RPI2, personally I use it for emergency connections to servers at work.



## Connecting with Your Desktop or Laptop

The first thing you need to connect to the RPI2 is its IP address. You use a variety of methods; check your router to see what devices are attached, use an IP scanning tool, or at the command line use a network discovery tool (nmap etc.). Here we use an IP scanning program (several are available). In this example “Advanced IP Scanner” is used on Windows 10:



In this example the RPI2 is visible both by Ethernet and Wi-Fi:

* **192.168.2.11** – wired address
* **192.168.1.2** – Wi-Fi address

Your addresses will vary depending on your networking setup and how you have the RPI2 connected**. If the neither interface is showing up, don’t panic! Just connect to the RPI2 with an HDMI cable and USB keyboard.** If you can find an IP address though, you then have the option of working entirely remotely on your desktop or laptop.

Once you know where to connect to you can use any tool that supports ssh style connections:

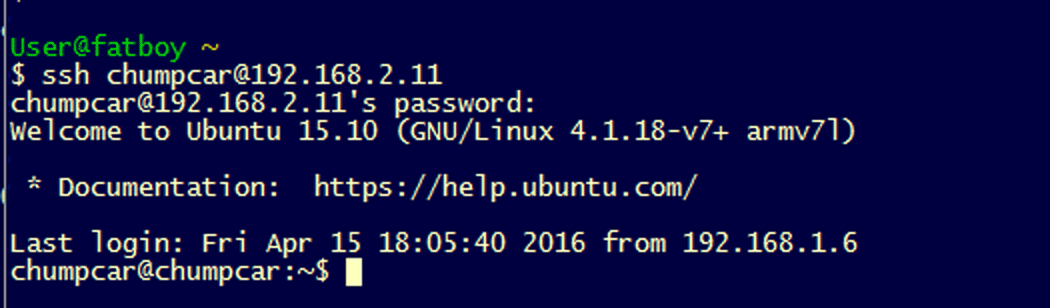
**PuTTY** - <http://www.chiark.greenend.org.uk/~sgtatham/putty/>

**ssh** (Cygwin) - <https://www.cygwin.com/>

**Some versions of Windows** support ssh - <http://www.powershelladmin.com/wiki/SSH_from_PowerShell_using_the_SSH.NET_library>

**Eclipse (RSE)** - <http://www.patrickjwaters.com/blog/2011-07-24/how-setup-eclipse-php-pdt-remote-system-explorer-theme-manager-and-drupal-plugins/35>

Personally, I favor Cygwin as I use it for all my server/web work already on my Windows 10 home desktop.



# System Setup

## OS Install

The Raspberry PI 2 (RPI2) has been loaded with “Ubuntu Mate” for the Operating System (OS):

<https://ubuntu-mate.org/raspberry-pi/> (don’t forget to resize the flash card!)

Several distributions are available for the RPI2, but most of them come pre-loaded with some variety of “E-Z” use OS. You then have to work within whatever E-Z framework is provided, rather than having the full power of Linux. With Ubuntu Mate…there is not fence; you can roam as you please.

However this flexibility comes at a small price; Ubuntu Mate is “headless” it doesn’t have an SSH style setup on first boot; it requires you to use its GUI to walk through a setup wizard on screen. This isn’t such a big limitation though; plugin a USB keyboard/mouse (if you are using a KBI just add it to one of your slots), and hook up an HDMI cable from the RPI2 to your display. If you have a DVI or VGA based display, you’ll need to dig out a conversion cable.

Once you are done the initial setup though, you need never use the GUI again. You can now happily use SSH via Ethernet or WIFI. To use WI-FI you’ll need to plugin the WI-FI adaptor. (see the section below).

Beyond the standard setup, configure the system to your preference. Add appropriate user accounts, and using apt-get (not Yum, this is Ubuntu not CentOS!), install your favorite packages.

## Real Time Clock (RTC)

To provide a consistent definition of time and time stamps, even when NTP isn’t possible (i.e. Not connected to the net), we use the PIFACE RTC module:

<http://www.piface.org.uk/products/piface_clock/>

This is an i2c module, so it doesn’t consume GPIO pints, its simply and add on for the i2c bus. Configuration is relatively easy, add the following to /etc/rc.local:

modprobe i2c-dev

i2cset -y 1 0x6f 0x08 0x47

modprobe i2c:mcp7941x

echo mcp7941x 0x6f > /sys/class/i2c-adapter/i2c-1/new\_device

( sleep 2; hwclock -s ) &

Not that some distributions (Ubuntu Mate) have a corrupted rc.local file; the first line has a tab before the # which stops it from running as a shell script. Just delete the tab.

You can manually check the hardware clock:

chumpcar@chumpcar:~$ sudo i2cdetect -y 1

0 1 2 3 4 5 6 7 8 9 a b c d e f

00: -- -- -- -- -- -- -- -- -- -- -- -- --

10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --

20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --

30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --

40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --

50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --

60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- UU

70: -- -- -- -- -- -- -- --

chumpcar@chumpcar:~$

If you see at 60:f the value 6F instead of UU, that means the kernel module hasn’t been added yet. You need to do the above echo of mcp7941x but be sure to do it while in a shell that is **already** at root level.

You can easily check that the RTC is working with:

chumpcar@chumpcar:~$ sudo hwclock -r

Thu 14 Apr 2016 09:11:24 PM EDT .105160 seconds

Also that it is configured properly on boot:

dmesg

…

[ 7.261940] rtc-ds1307 1-006f: rtc core: registered mcp7941x as rtc0

[ 7.261999] rtc-ds1307 1-006f: 64 bytes nvram

[ 7.262054] i2c i2c-1: new\_device: Instantiated device mcp7941x at 0x6f

## FTDI RS-232 Quad Conversion Cable (USB)

To connect the Raspberry PI 2 (RPI2) to the SoloDL and the DL-32, we use simple RS-232 connections. There is no modem control line usage etc, its straight RX/TX wires from RS-232 cables to our conversion cable (which then connects to the RPI2). The conversion cable is USB cable that ends with 4 D-Sub connectors that are wires for RS-232. The USB cable has a built in FTDI chipset that maps the 4 cables to 4 USB/Serial ports.

These 4 ports are automatically recognized by Linux and standard drivers. FTDI chips are well known to Linux. So nothing special for configuration is needed. Just plugin in the cable. Run dmesg to see the status:

dmesg

…

[ 4.745141] usbcore: registered new interface driver usbserial

[ 4.745279] usbcore: registered new interface driver usbserial\_generic

[ 4.745397] usbserial: USB Serial support registered for generic

[ 4.768727] usbcore: registered new interface driver ftdi\_sio

[ 4.773685] usbserial: USB Serial support registered for FTDI USB Serial Device

[ 4.774064] ftdi\_sio 1-1.3:1.0: FTDI USB Serial Device converter detected

[ 4.774303] usb 1-1.3: Detected FT4232H

[ 4.781836] usb 1-1.3: FTDI USB Serial Device converter now attached to **ttyUSB0**

[ 4.782673] ftdi\_sio 1-1.3:1.1: FTDI USB Serial Device converter detected

[ 4.783049] usb 1-1.3: Detected FT4232H

[ 4.800107] usb 1-1.3: FTDI USB Serial Device converter now attached to ttyUSB1

[ 4.800286] ftdi\_sio 1-1.3:1.2: FTDI USB Serial Device converter detected

[ 4.800514] usb 1-1.3: Detected FT4232H

[ 4.801332] usb 1-1.3: FTDI USB Serial Device converter now attached to **ttyUSB2**

[ 4.801479] ftdi\_sio 1-1.3:1.3: FTDI USB Serial Device converter detected

[ 4.801700] usb 1-1.3: Detected FT4232H

[ 4.802749] usb 1-1.3: FTDI USB Serial Device converter now attached to **ttyUSB3**

The /dev/ttyUSB<X> devices are dependent on run-time resources and kernel errors etc, so the values for <X> will change, but the ordering doesn’t. If you look closely at the quad cable you’ll see numbers printed on the D-Sub connectors. Those numbers represent the order the cable ends points will be recognized in.

To simply things further I’ve made the D-sub connectors:

* **RED** – for the DL-32
* **YELLOW** – for the SoloDL

Internally the DL-32 is the 1st cable, and the SoloDL is the 2nd cable. The ECU Bridge will automatically detect the order of the cables, and when they are connected or disconnected…regardless of what the actual /dev/ttyUSB<X> name is.

## Apache / PHP

Setting up Apache/PHP (so we can have web based GUI for the ECU bridge) requires just a few package installs. There is a walk through over here:

<https://www.howtoforge.com/ubuntu-lamp-server-with-apache2-php5-mysql-on-14.04-lts>

The short of it:

sudo apt-get install apache2

sudo apt-get install php5 libapache2-mod-php5

sudo apt-get install php5-ssh2 php5-mcrypt php-http-request2 php-doc php5-xmlrpc php5-readline php5-pgsql php5-mysql php5-ldap php5-json php5-gd php5-curl php5-cli

service apache2 restart

Follow the steps for testing the default web site and PHP. The key thing is to confirm that things like JSON and cURL support are available in PHP (to support a REST API into the Raspberry).

## WI-FI Adaptor (USB)

To allow connection to a phone or laptop without having an Ethernet connection, the Wi-Pi dongle can be use used:

<https://www.element14.com/community/docs/DOC-69361/l/wifi-usb-dongle-for-raspberry-pi>

The chipset is supported by the standard drivers included by most Linux distributions, including Ubuntu Mate, and Raspbian even supports it out of the box. Unfortunately the vendor (Element 14) only provides installation instructions for Raspian, and Ubuntu Mate isn’t setup by default to be configured for Wi-Fi, nor have a appropriate Wi-Fi network address , SSID etc.

These details can be updated manually though, and even if the Wi Fi interface does not come up automatically on reboot of the Raspberry PI you can connect with Ethernet and use ssh/putty to manually edit the necessary system files and “bounce” the networking service etc.

There are only a few critical system files you’ll need to update, one is the /etc/network/interfaces file:

…

auto wlx40a5ef061938

iface wlx40a5ef061938 inet dhcp

wpa-conf /etc/wpa\_supplicant/wpa\_supplicant.conf

address 192.168.1.2

netmask 255.255.255.0

network 192.168.1.0

broadcast 192.168.1.255

gateway 192.168.1.1

Notice that instead of “wlan0” the interface name “wlx40a5ef061938” is used. This is (for whatever reason) what the Linux kernel will use to identify the USB based Wi-Fi adaptor. I could likely be renamed, but I haven’t bothered with that detail. For now, just use the auto-detected interface name.

For netmask/network/gateway/address/broadcast values, these will all be particular to your Wi-Fi network or hotspot for your phone/laptop. When I tested with my router (192.168.1.1), I had to give it an appropriate static IP that isn’t used yet. It may be possible to have both Ethernet and Wi-Fi interfaces active on the Raspberry at the same time with DHCP assigned IP addresses...but for now its easier to simply pick an address and go. An address is needed to separate Ethernet from Wi-Fi…otherwise you can only use one or the other at the same time…which makes it hard to test Wi-Fi if you don’t have the RPI2 hooked up to a monitor and USB keyboard.

To provide your Wi-FI password or other settings, you can update

/etc/wpa\_supplicant/wpa\_supplicant.conf:

network={

ssid="NETGEAR25"

psk="xxxxxxxx"

key\_mgmt=WPA-PSK

}

Obviously fill in values appropriate for you Wi-Fi network or hotspot.

Finally you must update the service configuration for the wpa\_supplicant daemon /lib/systemd/system/wpa\_supplicant.service:

[Unit]

Description=WPA supplicant

Before=network.target

[Service]

Type=dbus

BusName=fi.epitest.hostap.WPASupplicant

ExecStart=/sbin/wpa\_supplicant -u -s -dd -t -B -P /run/wpa\_supplicant.wlx40a5ef061938.pid -i wlx40a5ef061938 -D nl80211 -c /etc/wpa\_supplicant/wpa\_supplicant.conf

[Install]

WantedBy=multi-user.target

Alias=dbus-fi.epitest.hostap.WPASupplicant.service

Notice that its been updated to use the auto-detected interface name and –dd was added to provide extra debugging in the /var/log/syslog log file. The nl80211 driver is a “mother of all drivers” that handles various chipsets. It’s the one used by modern versions of Linux to talk to Wi-Fi adaptors.

To actually make sure the appropriate services are running, should kick “networking”, and “wpa\_supplicant”. If the interface still isn’t coming up, you can use ”ifup” to manually force it up. The wpa\_supplicant daemon is a bridge used to enforce WPA encryption details, which is required for access to most Wi-Fi networks. So its not simply a matter of ensuring the networking is running and the kernel has detected the Wi-Pi driver…wpa\_supplicant must all be successfully running.

# make sure the actual interface (wlan0 a.k.a. wlx40a5ef061938) is configured:

sudo service networking restart

# make sure the WPA supplicant daemon is running:

sudo systemctl start wpa\_supplicant

# you can use ‘status’ to see if its already running, and restart to bounce it.

# if the interface doesn’t come up, you can force it up:

sudo ifup wlx40a5ef061938

# for everything, check /var/log/syslog if you suspect problems. I’ve enabled

# extra debugging (-dd) on the supplicant daemon, so the log will have lots of

# details if you need it.

# check the interfaces:

chumpcar@chumpcar:~$ ifconfig

enxb827ebbb5c90 Link encap:Ethernet HWaddr b8:27:eb:bb:5c:90

inet addr:**192.168.2.11** Bcast:192.168.2.255 Mask:255.255.255.0

inet6 addr: fe80::ba27:ebff:febb:5c90/64 Scope:Link

UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1

RX packets:14308 errors:0 dropped:0 overruns:0 frame:0

TX packets:3830 errors:0 dropped:0 overruns:0 carrier:0

collisions:0 txqueuelen:1000

RX bytes:755549 (755.5 KB) TX bytes:771485 (771.4 KB)

lo Link encap:Local Loopback

inet addr:127.0.0.1 Mask:255.0.0.0

inet6 addr: ::1/128 Scope:Host

UP LOOPBACK RUNNING MTU:65536 Metric:1

RX packets:302600 errors:0 dropped:0 overruns:0 frame:0

TX packets:302600 errors:0 dropped:0 overruns:0 carrier:0

collisions:0 txqueuelen:0

RX bytes:28749263 (28.7 MB) TX bytes:28749263 (28.7 MB)

wlx40a5ef061938 Link encap:Ethernet HWaddr 40:a5:ef:06:19:38

inet addr:**192.168.1.2** Bcast:192.168.1.255 Mask:255.255.255.0

inet6 addr: fe80::42a5:efff:fe06:1938/64 Scope:Link

UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1

RX packets:44860 errors:0 dropped:0 overruns:0 frame:0

TX packets:1497 errors:0 dropped:0 overruns:0 carrier:0

collisions:0 txqueuelen:1000

RX bytes:5252032 (5.2 MB) TX bytes:174478 (174.4 KB)

chumpcar@chumpcar:~$

Once your Wi-FI interface is up, you can use commands like “iwlist” to do handy things like look for the networks your Wi-Fi adaptor can see:

chumpcar@chumpcar:~$ sudo iwlist wlx40a5ef061938 scan | head -10

wlx40a5ef061938 Scan completed :

Cell 01 - Address: 02:15:99:0D:87:86

Channel:8

Frequency:2.447 GHz (Channel 8)

Quality=59/70 Signal level=-51 dBm

Encryption key:on

ESSID:"**DIRECT-wxC410 Series**"

Bit Rates:6 Mb/s; 9 Mb/s; 12 Mb/s; 18 Mb/s; 24 Mb/s

36 Mb/s; 48 Mb/s; 54 Mb/s

Mode:Master

chumpcar@chumpcar:~$

If things go badly, and both Ethernet and Wi-Fi interfaces become unresponsive, and you can’t connect with ssh or putty…no worries. Just plugin an HDMI monitor and USB keyboard. You aren’t locked out. You may however have to sweat through some re-configuration.

By default the Ethernet interface is the primary interface, so if you have any issues with Wi-Fi access, plugin an Ethernet cable and ssh over the RPI2 and run ifup force the Wi-Fi connection online.

## ECU Bridge

The ECU bridge is the software that we have implemented to take data from the DL-32 (an analog to digital converter data logger), and pass it over to the Solo DL (a digital data logger). The RPI2 is basically being used as a fancy patch cable to go from one to the other. The ultimate goal is get data from the Honda Engine Control Unit (ECU) to the DL-32 and then on through to the Solo DL and for logging, analysis and live feed to the Smarty Cam for the driver to view in real-time.

There are three components to the ECU bridge:

**ecubridge** – this daemon is the main show. It interfaces with the DL-32 and the SoloDL to actually bridge them together. At the same time, it pass data to the ecudatalogger (below) to do onboard logging of the data that gets processed (for after action analysis or simulation in later coding we might do).

**ecudatalogger** – this daemon simply listens to any of three outputs from the ecubridge (raw data from the DL-32, processed – “normal” data, or the final output data to the SoloDL), and then passes any data captured to RRD for storage. This is how we implement on board data logging. In addition, RRD givers us all the usual abilities of RRD, high speed logging, graphing and analysis of the logged data etc.

**rrdcached** – this is the standard RRD daemon used to allow a client (ecudatalogger) to log data to memory (quickly) and then at appropriate intervals write the data to the RRD data files on disk (flash memory).

The three daemons are setup to run automatically at boot time, and don’t require any user intervention. Basically the RPI2 will operate as a consumer device to bridge the DL-32 to SoloDL. Just make sure the cables are plugged in. ☺

A note on daemon startup order; there is no dependency; The bridge uses UDP multi-cast to chat with the logger, so the order which one starts doesn’t matter. The logger and RRD also don’t have an ordering; the logger is smart enough to just wait if RRD isn’t up yet. You only need to power on RPI2 and the rest is automatic. Even if you yank out the USB cable or haven’t plugged it in yet…the bridge will auto-detect and take appropriate action. Obviously the data recorded while USB cable is unplugged will be null.

### Installing ecubridge

The source code is stored in ~/dev/chumpcar (or wherever you checked out the code form github). Change directory to the top level of the source folder and do:

make clean daemon

This will give you a fresh build of the ecubridge daemon. You can then do:

sudo make install

to get it installed. Key system files you should know about:

/var/log/ecubridge – the log file

/etc/ecubridge/ecubridge.ini – the configuration file

/var/run/ecubridge.pid – the pid/lock file

/usr/local/bin/ecubridge – the actual binary

/etc/init.d/ecubridge – the start/stop script

Normally the most you should have to do is ensure the new ecubridge executable gets copied to /usr/local/bin/ecubridge. If the service is already running, you have to do

Sudo service ecubridge stop

To stop it first. You can use start to “start” it up again. You can use status to check if its live:

chumpcar@chumpcar:~$ service ecubridge status

â— ecubridge.service - (null)

Loaded: loaded (/etc/init.d/ecubridge)

Active: active (running) since Fri 2016-04-15 17:53:44 EDT; 3h 10min ago

Docs: man:systemd-sysv-generator(8)

CGroup: /system.slice/ecubridge.service

â””â”€798 /usr/local/bin/ecubridge

Apr 15 17:53:43 chumpcar systemd[1]: Starting (null)...

Apr 15 17:53:43 chumpcar ecubridge[788]: Starting ECU Bridge...

Apr 15 17:53:44 chumpcar ecubridge[788]: Started.

Apr 15 17:53:44 chumpcar systemd[1]: Started (null).

chumpcar@chumpcar:~$

With the ECU Bridge daemon started, you should now have data flowing to the Solo DL. If the DL-32 is powered, the data will not be “0” it will be whatever is coming from the DL-32 (at least on the first 5 channels). You can easily check this by opening Race Studio 2 on you computer and opening the “Online” dialog. As long as the Solo DL is connected to your computer with a USB cable…you should see either 0 or some positive value in all 15 channels. If you see “ERR” for channel values that means there is a disconnect somewhere or a bug in the ECU Bridge; the Solo DL is not receiving data.

### Installing ecudatalogger

Similar to bridge, the logger can be built with:

make clean logger

and then installed with:

sudo make install

The logger is similarly controlled via the service command. Key files for the ECU data logger:

/var/log/ecudatalogger – the log file

/etc/ecubridge/ecudatalogger.ini – the configuration file

/var/run/ecudatalogger.pid – the pid/lock file

/usr/local/bin/ecudatalogger – the actual binary

/etc/init.d/ecudatalogger – the start/stop script

### Installing rrdcached

This is a standard system package so not much is needed here beyond simply running apt-get install … if its not already installed. Keep in mind though the RRD data files we use are stored here:

$ ls –l /var/lib/rrdcached/db/

-rw-r--r-- 1 root root 51850188 Apr 15 13:33 ecu-data-normal.rrd

-rw-r--r-- 1 root root 51850188 Apr 15 13:33 ecu-data-output.rrd

-rw-r--r-- 1 root root 51850188 Apr 15 13:33 ecu-data-raw.rrd

Each one corresponds to the kind of date we log; raw is straight from the DL-32, normal is the processed data – but before transforming to SoloDL output, and finally output is the final data that is sent to the Solo DL – raw but for the Solo DL instead of the DL-32. Each RRD file is setup to be a big ring buffer for 24 hours worth of data at a resolution of 1 sec (the finest resolution supported by RRD).

If you want to clear the data logs, just remove them.

Because data logging is continuous, after completing a race you should POWER OFF the RPI2, otherwise it will keep recording data (empty data) and overwrite any valuable data logged during the race.

Alternatively, after a race make a copy of the data files and just keep them in an archive somewhere. Currently the RRD files are about 51MB each (150MB total). Obviously there is room to spare, so we could in theory make these much bigger. However, 24 hours should be enough to capture most races we expect the car to be in.

If you just want to quickly see what the last 30sec of data logged is, you can run a command like this:

rrdtool fetch \

--daemon unix:/var/run/rrdcached.sock \

/var/lib/rrdcached/db/ecu-data-output.rrd \

AVERAGE -r 1 --start -30

The ‘fetch’ command is to grab data from an RRD file (in ourcase the SoloDl output data), the daemon option is required because the data may not actually be written to file yet (flushed), you should read it live from the RRD daemon. Finally, the AVERAGE category tells us what kind of data to read, you can also choose MIN, MAX and LAST. The –start option can be used to indicate where to start reeling data from. This example we start from 30seconds ago.

If you see nothing but “nan”…no data is being logged, you should check the ecu bridge and data logger log files for problems.

You can find out more about the world of RRD here:

<http://oss.oetiker.ch/rrdtool/>