Ullr

Release 0.1.17

Zach Henry

Aug 25, 2022

GETTING STARTED

A serial <-> MQTT interface for sports timing.

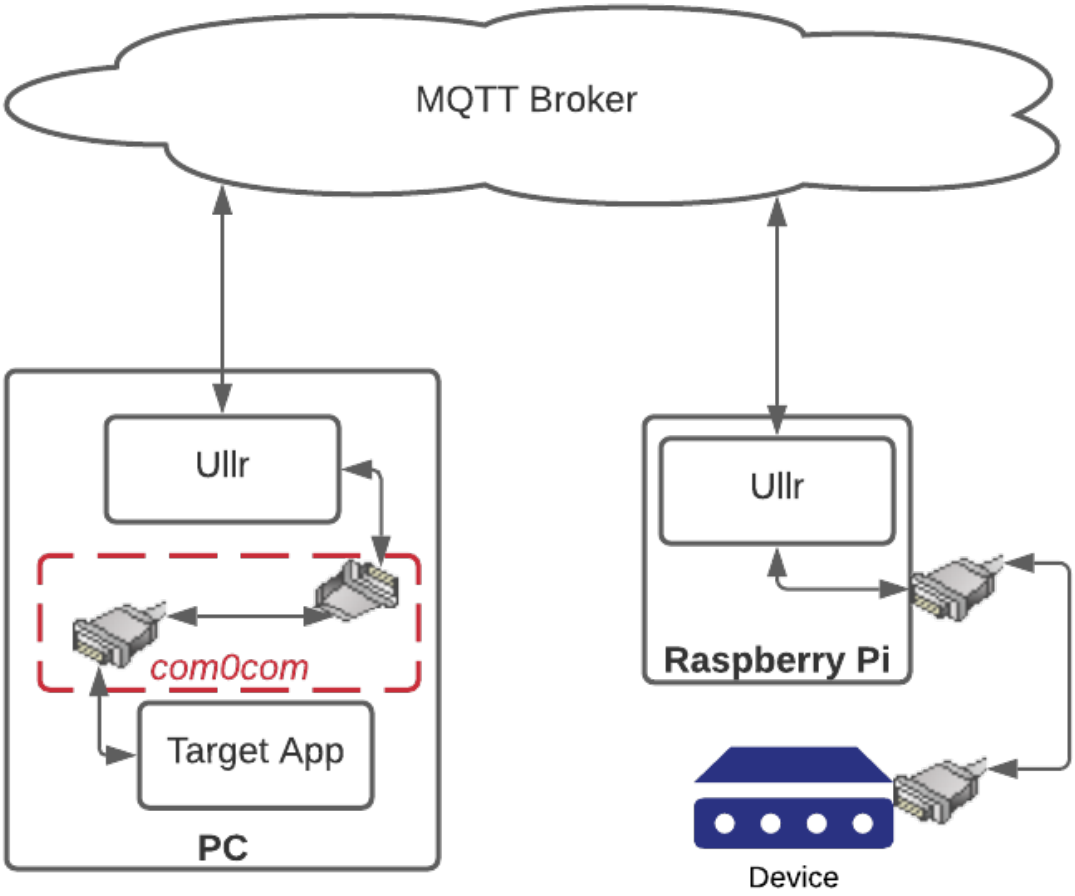

Fig. 1: Ullr signal flow.

Ullr (pronounce "OOH-ler") allows for reading and writing data from remote serial devices. It was designed with *ski racing* in mind, but can be used to access any remote serial device with an available internet connection. It also acts as a hub, allowing for an arbitrary number of serial devices to be connected to an arbitrary number of software instances.

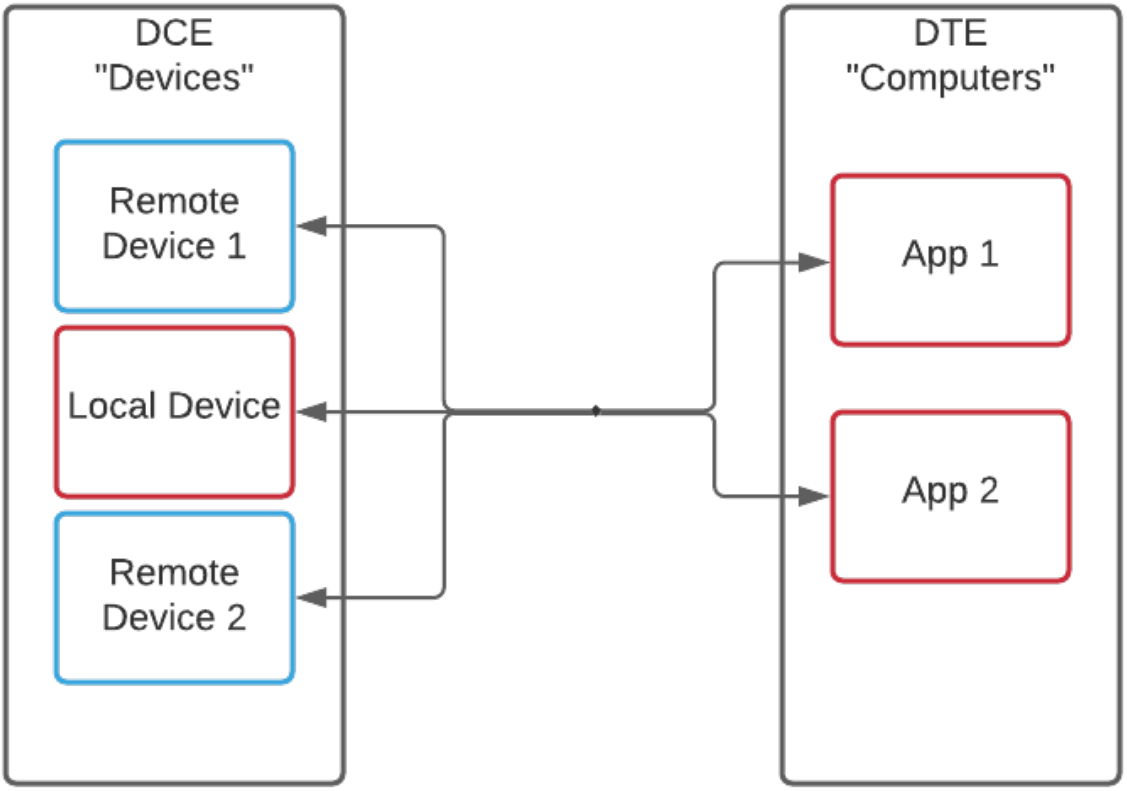

Fig. 2: Multiple devices sharing a single serial connection.

CHAPTER

ONE

INSTALLATION

```
pip install ullr
```

The simplest way to install Ullr is in a Python environment using pip. It is recommended, if possible, to install as superuser in Linux environments.

An installer for the latest Windows release can be found here: Latest Windows Release.

The latest Debian(amd64) binary can be found here: Latest Debian(amd64) Release.

An in-depth guide to installation on a Raspberry Pi can be found *here*.

CHAPTER

TWO

CONFIGURING ULLR

See the *Configuration* section for an in-depth guide to configuring Ullr.

CHAPTER

THREE

RUNNING

From a command line, type `ullr`. Or navigate to the Windows start shortcut. By default, Ullr starts a web interface on localhost:5000. Navigate to this page from a browser.

Fig. 1: The Ullr web interface.

See the *Usage* section for more information.

CHAPTER

FOUR

GET THE SOURCE

Ullr is published under the GNU GPL 3 license.

The source is always available on the Ullr GitHub homepage and available for download here.

4.1 RS-232 Terminology

Ullr borrows terminology from the original RS-232 specification, specifically the terms "DCE" and "DTE". "DCE" stands for Data Circuit-terminating Equipment. These are the serial devices, such as timers, sensors, and thermometers. "DTE" stands for Data Terminal Equipment. These are the computers and software instances. The RS-232 protocol is how DCE devices communicate with DTE devices over a serial cable.

Serial cables have a major drawback: they are only good for ten meters or so. Ullr solves this problem by tranporting serial messages with the MQTT protocol.

4.2 MQTT Messaging Protocol

Ullr makes use of the MQTT messaging protocol to send and receive remote serial messages. This protocol was originally developed to monitor remote oil pipelines with satellite connections, and is particularly sturdy in situations with unreliable internet connections. It is also lightweight, and doesn't require much data or processing power.

MQTT is based on a publisher/subscriber model. Devices are NOT connected directly to each other, but to a central hub called a broker. When a device has information to send, it publishes it to the broker along with a topic name. The broker then sends this message to all devices that are subscribed to the topic. The protocol makes it possible to guarantee that all published messages get delivered to all subscribed clients at least once. For an excellent in-depth description of the MQTT protocol, see the HiveMQ MQTT Essentials Guide.

Ullr makes use of this protocol by allowing local serial devices to be published. Ticking the "Published" checkbox on a local device will send all messages from the device to the MQTT broker. The topic the messages are published to will be the MAC address of the host client with the colons removed, followed by a forward slash and the device name with the spaces replaced by underscores. For example, for a device named "Thermometer" on a computer with a MAC address of 00:16:3e:2b:2f:28, messages will be published to the topic "00163e2b2f28/Thermometer". This topic name can be seen by clicking the hamburger icon in the bottom left corner of any device on the WebUI. If we want to connect to this device using Ullr on a remote PC, we can use the WebUI to add a remote device subscribed to "00163e2b2f28/Ullr".

4.3 Virtual Null Modems

A null modem, in the physical world, is just a serial cable used to connect two DTE devices. A virtual null modem is the software equivalent of the same thing. You can think of it as two com ports connected by a serial cable: what is sent to one com port is received by the other, and vice versa.

What do virtual null modems have to do with Ullr? A virtual null modem is what we need to connect Ullr to another piece of software. It is the pipe that carries serial data from Ullr to the target software. Ullr is connected to one end of the virtual null modem, and the target software to the other.

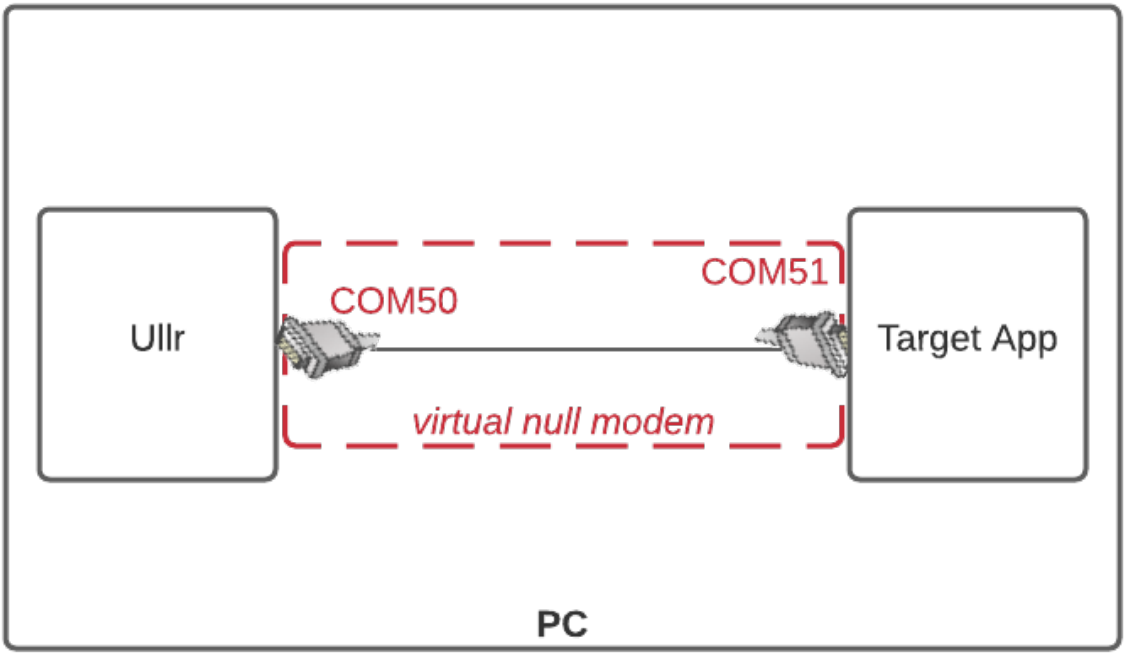

Fig. 1: Diagram of virtual null modem functionality.

In practice, a virtual null modem just appears as two available serial ports on your machine. In the example above they would be ports COM50 and COM51. We'll go through setting it up on various operating systems below.

4.3.1 Windows

There are multiple virtual null modem products available for Windows, but we'll focus on com0com, as it is stable as well as free and open source. There is a signed Windows installer available from Alge at com0com (signed). Using the signed installer avoids possible permission and security issues with Windows.

Once com0com is installed, it is necessary to open the configuration program from the start menu and add a linked pair of com ports (a virtual null modem). You can choose any names you want for these ports, as long as they aren't already in use. It is best practice, though not strictly necessary, to start the name with COM (for example, COM21 or COM50). This is because not all software will recognize com ports that don't follow this convention.

Running the com0com configuration tool will install drivers for the null modem on your machine. This null modem will persist until the drivers are removed, and there is no need to run this setup again.

4.3.2 Linux

On Linux (or other POSIX-like systems) a null modem can be setup much more simply by using socat. Make sure socat is installed on your system, then type:

```
sudo socat PTY,link=/dev/ttyS50 PTY,link=/dev/ttyS51
```

This example creates a virtual null modem by linking the ports /dev/ttyS50 and /dev/ttyS51. Note that you can choose any name you'd like for the ports, but starting the name with 'ttyS' will increase the chances of the ports being detected by Ullr and other software.

Unlike the Windows setup above, this virtual null modem will only persist as long as socat is running. If you'd like to run socat in the background, you can add an ampersand to the end of the command, like so:

```
sudo socat PTY,link=/dev/ttyS50 PTY,link=/dev/ttyS51 &
```

If you'd like socat to run on boot you could add the following line to your cron table:

```
@reboot socat PTY,link=/dev/ttyS50 PTY,link=/dev/ttyS51
```

See *Using cron* for more detail on editing the cron table.

4.4 Running on a Raspberry Pi

Ullr was designed to be lightweight and useful in portable, outdoor applications. The Raspberry Pi is a great computer for these needs. This section will provide some in-depth info on installation, operation and configuration on a Raspberry Pi, specifically a Raspberry Pi Zero W.

4.4.1 Choosing an OS

There are a number of Raspberry Pi operating systems to choose from. Virtually all of them are based on the Linux Kernel, and are tailored to suit different needs. Some offer a basic Graphical User Interface, while others are "headless", meaning they offer a Command Line Interface only. Either will work, but this guide recommends the use of Raspberry Pi OS Lite, a headless OS. This saves on processor and battery use and reduces complexity somewhat. The instructions in this section will be based on command line operation, but it is certainly possible to use a graphical interface as well.

4.4.2 Some Linux Basics

How to use Linux is mostly outside the scope of this guide. There are some excellent resources on the web. However, we'll go over a few essential concepts.

Superuser

Linux does not allow regular users to make system-wide changes by default. This is an important thing to keep in mind as we install and setup Ullr. The default user on a Raspberry Pi is "pi". This user does not have permission to install software or access serial ports. When we need to execute a command or run a program that does these things (like Ullr), we should put the "sudo" in front of the command. This will execute the command as as superuser with admin (root) priveleges. Keep in mind that whenver we use sudo we are acting as a different user, NOT the "pi" user. Changes we make will happen system wide.

4.4.3 Installing on a Raspberry Pi

Installing Python

The first step on a fresh Raspberry Pi OS install is to install python. We'll use apt, the built in package manager. On a command line, type:

```
sudo apt install python3 python3-pip
```

This will install the latest available version on Python, as well as pip, the Python package manager.

Installing Ullr

The next step is to install Ullr using pip. Type:

```
sudo pip3 install ullr
```

Ullr should now be downloaded and installed! To run, just type:

```
sudo ullr
```

You should get the following message in the terminal:

```
Ullr running on port 5000
```

Ullr is now up and running! A web configuration interface is available on port 5000 of the Raspberry Pi. If you're running a Raspberry Pi OS with a graphical interface, you can open a browser window to `localhost:5000`. This isn't the best way to access the interface, however, due to the web rendering limitations of the Raspberry Pi. A better way is to access the interface remotely from a computer on the same LAN. From the remote PC navigate to `raspberrypi.local:5000/`, or `<pi-ip-address>:5000/`.

Why sudo? Ullr requires access to serial ports, which by default are not accessible by regular users. If you don't want to run as superuser, make sure you run as a user with serial port permissions.

4.4.4 Setting Ullr to run on boot

Running Ullr from the command line is easy enough when we have a monitor and keyboard or remote access available, but that isn't always the case. We can make things easier by setting Ullr to start whenever the Raspberry Pi is powered on.

Using cron

The simplest way is to use cron, the Linux task scheduler. Before we start changing settings, we need to make sure the nano text editor is installed. Type:

```
sudo apt install nano
```

Once we're sure nano is there, the next step is to edit the cron table to schedule Ullr to run at boot. Type:

```
sudo crontab -e
```

There may be a prompt asking you to select a text editor. Choose nano. Next you will see a screen with a rather lengthy comment explanation. Below these comments, add the following line:

```
@reboot sudo ullr
```

The file should look similar to this when you're done:

```
  GNU nano 3.2                                          /tmp/crontab.SCcssK/crontab

# Edit this file to introduce tasks to be run by cron.
#
# Each task to run has to be defined through a single line
# indicating with different fields when the task will be run
# and what command to run for the task
#
# To define the time you can provide concrete values for
# minute (m), hour (h), day of month (dom), month (mon),
# and day of week (dow) or use '*' in these fields (for 'any').
#
# Notice that tasks will be started based on the cron's system
# daemon's notion of time and timezones.
#
# Output of the crontab jobs (including errors) is sent through
# email to the user the crontab file belongs to (unless redirected).
#
# For example, you can run a backup of all your user accounts
# at 5 a.m every week with:
# 0 5 * * 1 tar -zcf /var/backups/home.tgz /home/
#
# For more information see the manual pages of crontab(5) and cron(8)
#
# m h  dom mon dow   command

@reboot sudo ullr
```

Fig. 2: Cron table with ullr task added.

Hit control-x, y for yes, then enter to confirm. That's it! Ullr will now run whenever the Raspberry Pi is powered on.

Adding a systemd service

The system daemon (systemd) can run Ullr at boot just like cron. However, it has a few important advantages:

- Ullr is run as a service, that can be stopped, started or monitored at any time.
- Ullr will automatically restart in case of any failure.
- Ullr can be set to start only after other services, such as network connectivity, are functional.

The last point is particularly helpful for our use case. Ullr depends on an internet connection to make the initial connection to the MQTT broker. A connected network interface also makes determining the device's MAC address more reliable. Waiting to start Ullr until the network service is running will therefore help avoid any unexpected behavior.

To setup Ullr as a systemd service, we need to create a service file in the systemd directory. Open a blank file using nano:

```
sudo nano /etc/systemd/system/ullr.service
```

Then, copy and paste the following:

```
[Unit]
Description=Ullr Startup Service
After=network-online.target
Wants=network-online.target

[Service]
ExecStart=/usr/bin/python3 -m ullr
WorkingDirectory=/usr/bin
StandardOutput=inherit
StandardError=inherit
Restart=always
User=root

[Install]
WantedBy=multi-user.target
```

Exit nano and save the file. Now, we need systemd to reload our changes. Type:

```
sudo systemctl daemon-reload
```

Now we can start and stop Ullr as a service. Test it by typing:

```
sudo systemctl start ullr
```

and

```
sudo systemctl stop ullr
```

Once you're satisfied that the service runs correctly, all that's left is to enable it to run on boot.

```
sudo systemctl enable ullr
```

That's it! Ullr is now set to run on boot, after internet is connected, and restart in case of failure.

4.5 Configuration

Ullr functions just like a really long serial cable, to connect software and serial devices over long distances. Ullr needs to be installed and configured on both sides of the connection: on the computer running the target software, and on the computer connected to the serial device. Ullr can run on a laptop, a Raspberry Pi, or any device capable of running Python applications. See *Installation* for installation info.

The following is a guide for interactive configuration using the web interface. However, configuration can be done by simply *editing the config file*.

4.5.1 Navigating to the WebUI

Ullr hosts a web interface on port 5000 of the local machine by default. You can get there by typing `localhost:5000` in a browser window. If you used a Windows installer, there will also be a shortcut in the start menu.

Fig. 3: The Ullr web interface.

To get started, we need to setup Ullr for our needs. The configuration settings can be accessed by clicking the "Configure" button on the top right of the screen.

4.5.2 MQTT Broker Settings

The first step is to configure the MQTT broker. By default, these settings are set to the free shared Ullr broker.

However, Ullr can be configured to use any MQTT broker. Some free options are available at Hive MQ. For more information on how MQTT works, see *MQTT Messaging Protocol*.

4.5.3 Adding Local Devices

A local device is a device that is plugged into the local machine with a serial cable, or a piece of software running on the local machine. Click on the "Add Local" button to open the add device dialog.

Give the device a descriptive name. It can be anything that makes sense to you, like "CP540", "Start Timer", or "Split Second Software". Next, we need to tell Ullr whether this is a DCE or DTE device. Generally, if it's a physical device that's plugged into the computer is a DCE. If it's a piece of software it's a DTE.

The next step is to select the serial port and baudrate. This is the port the physical device is plugged into, or the port the software is listening on. Ullr automatically lists all available serial ports. If you don't see the port you're expecting, try refreshing the page or reconnecting your device.

Fig. 4: WebUI with configuration menu open.

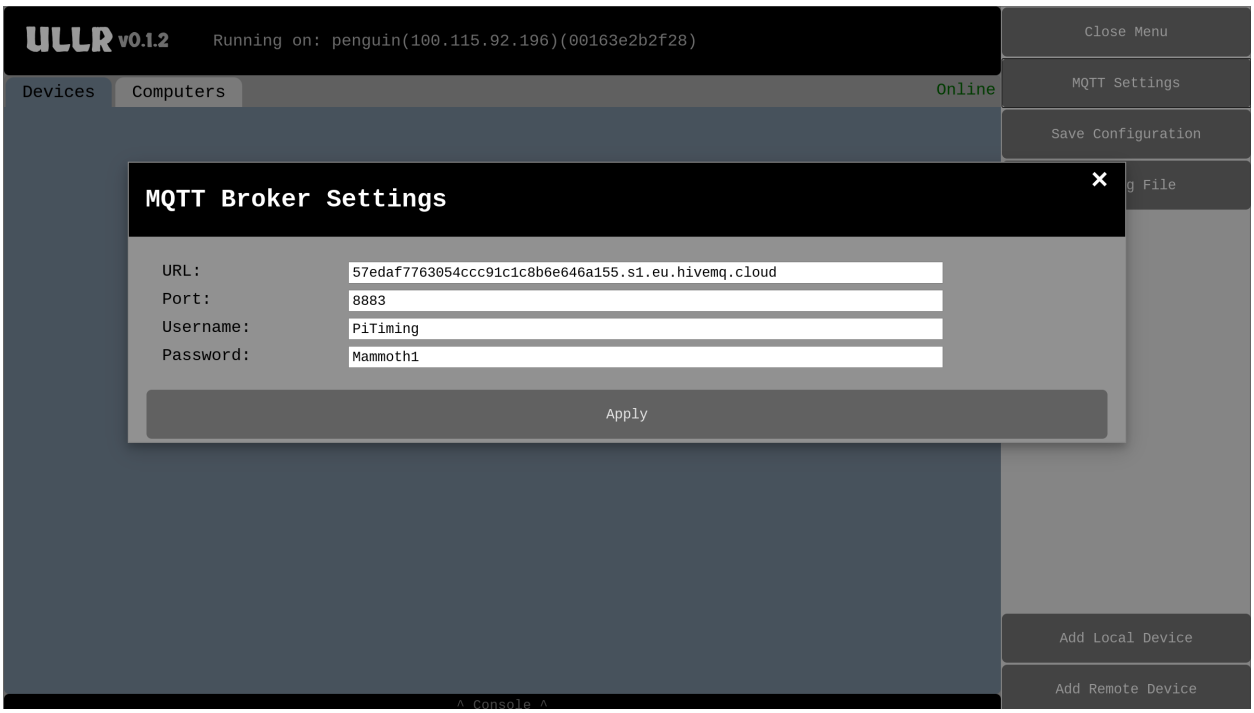

Fig. 5: The MQTT settings dialog.

Fig. 6: The local device dialog.

Finally, we have some choices to make reflected by the three checkboxes at the bottom. We can choose to Mute the device if we only want to send the device messages and don't want or expect to receive messages from it. Checking this box will reduce CPU usage for devices that don't send messages. By default this box is unchecked.

Along these same lines, we can choose whether or not the device accepts incoming messages. If we only expect the device to send messages, not receive them, this box can be unchecked. This will also reduce CPU usage. By default the box is checked.

Finally, we can choose whether or not the device is published. If a device is published, any message it sends will also be sent to the MQTT broker (the "cloud"), where it can be read remotely.

Click the red "Add" button to add the device. It will then appear in the device window under the appropriate tab: "Devices" if DCE, and "Computers" if DTE.

Clicking the hamburger icon in the lower left will bring up some advanced options for the device, which will be covered elsewhere in this document. At the top of the window is the "Published Name". This is what is necessary to connect to the device remotely. Clicking on the "X" in the lower right of the device will remove it. If you run into trouble, check the *console* for more information.

4.5.4 Adding Remote Devices

A remote device is something that's plugged into a remote computer. We'll use Ullr and the *MQTT Messaging Protocol* to access it. Click on the "Add Remote Device" button in the "Configure" menu to bring up the dialog.

Give the device a descriptive name. It could be "Split Timer", "Remote Sensor", "Display Board", or anything else that makes sense. The next field is labeled "Host ID/Device ID", and is the specific MQTT topic we are subscribing to. The Host ID will be the unique MAC address of the remote host with colons removed, and the Device ID will be the name of the remote device with spaces replace by underscores. For example, maybe we want to connect to a device called "CP 540" on a remote Raspberry Pi. If the Raspberry Pi's MAC address is "00:16:3e:2b:2f:28", we need to

Fig. 7: Local device added.

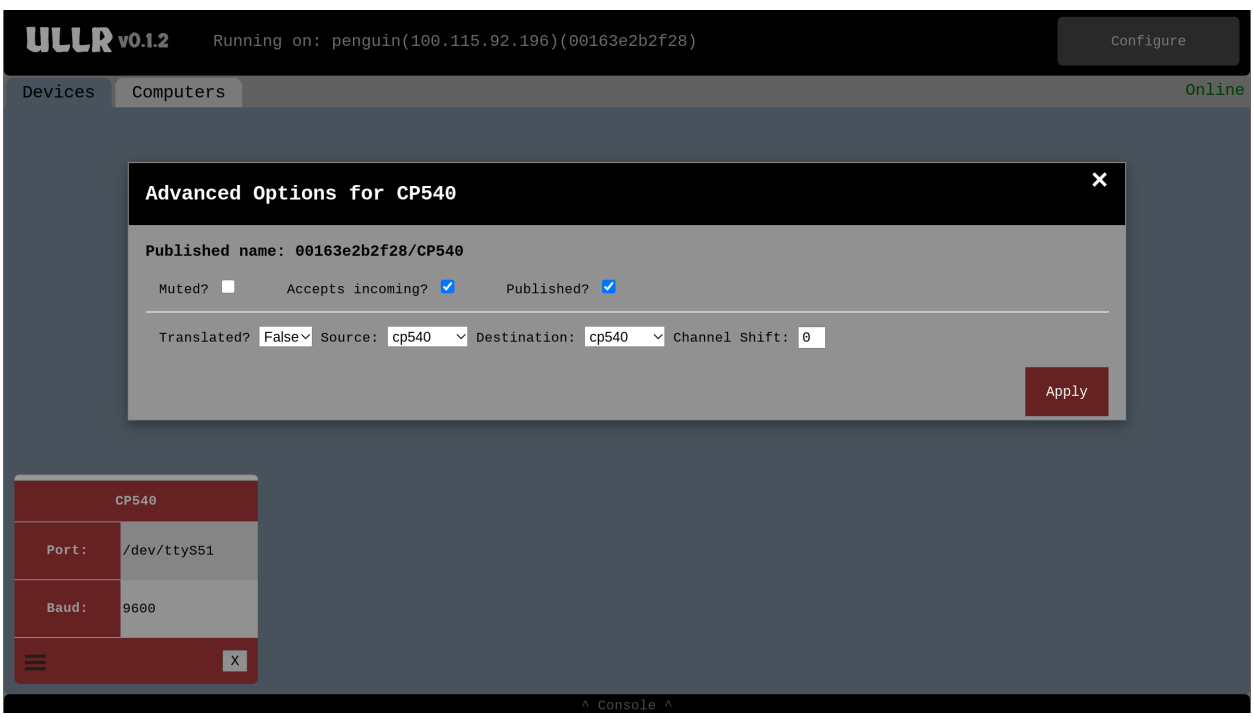

Fig. 8: Local device advanced settings.

Fig. 9: The remote device dialog.

enter "00163e2b2f28/CP_540" in the "Host ID/Device ID" field. If we are unsure of the device name, we can just enter "00163e2b2f28" to subscribe to ALL devices from the remote host.

Next is a field to determine how late messages are handled. When set to 0, Ullr will accept all messages regardless of how long they spend in transit. Otherwise, Ullr will only accept messages that arrive quicker than the value set in this field. Messages that arrive after the On-time Arrival window can then be dealt with manually. This is described in detail in the *Handling Late Messages* section.

The bottom 3 fields are identical to those described above in the *Adding Local Devices* section.

Click the blue "Add" button to add the device. It will then appear in the device window under the appropriate tab: "Devices" if DCE, and "Computers" if DTE.

Clicking the hamburger icon in the lower left will bring up some advanced options for the device, which will be covered elsewhere in this document. Clicking the "X" in the bottom right will remove the device. If you run into trouble, check the *console* for more information.

Fig. 10: Remote device added.

4.5.5 Saving configuration

Once the configuration is complete, you should save the settings to save time the next time Ullr is run. Click the "Save Configuration" button in the "Configure" menu. The configuration will be saved to your home folder. On Windows machines this might be C:Usersjdoe.configullrconfig.ini. On linux machines it might be /home/jdoe/.config/ullr/config.ini, or /etc/ullr/config.ini if run as Superuser.

4.6 Usage

Ullr facilitates the connection of DCE and DTE devices (see *RS-232 Terminology*). Any message from a DCE device is sent to all DTE devices, and any message from a DTE device is sent to all DCE devices. See the diagram below:

In this way it is possible to access a device such as a GPS receiver from multiple remote computers, as well as the computer the device is actually plugged into. It is also possible to connecte multiple devices to one piece of software as if they were a single device.

Once Ullr is configured and running (see *Configuration*) it is not necessary to use the web interface. It is sufficient to just run your target software, and Ullr will work in the background. However, the web interface provides some powerful features for monitoring operation and handling exceptions.

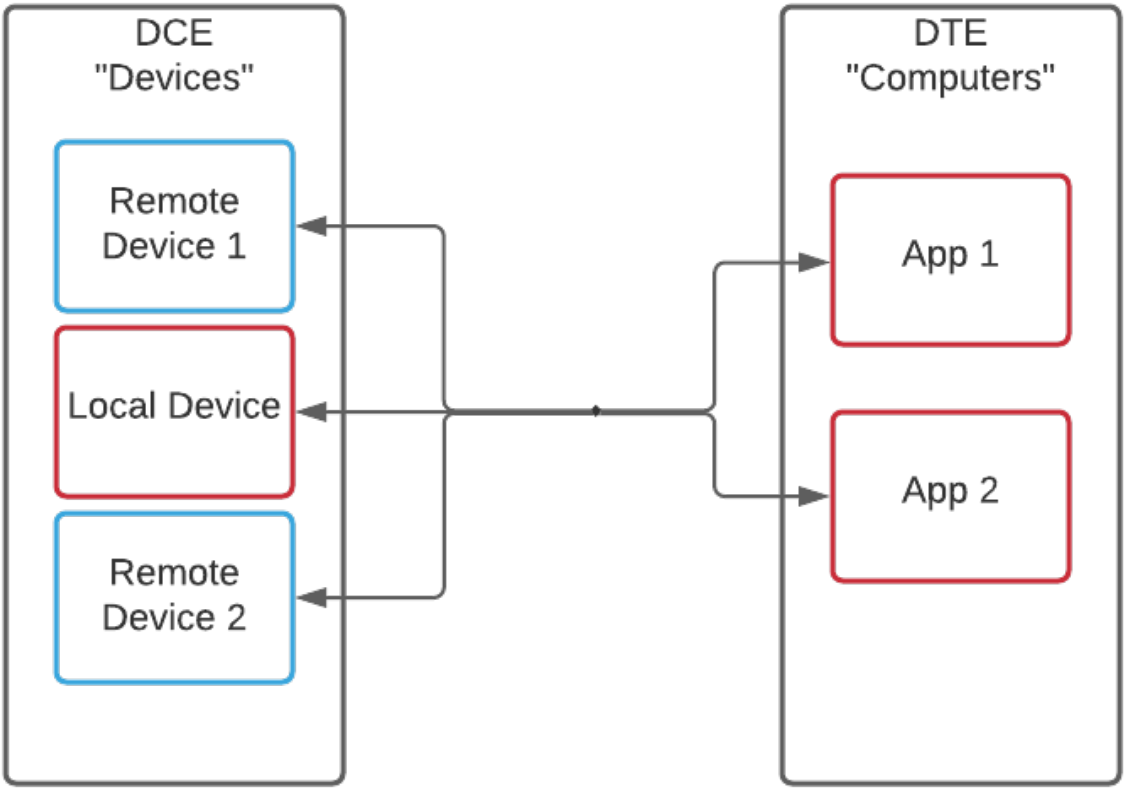

Fig. 11: Signal flow between devices in Ullr.

4.6.1 Monitoring Message Transmission

For this guide, we'll assume that you have added a few devices in *the previous step*. For example, you might have a remote DCE device, a local DCE device, and a local DTE software instance as in the example below.

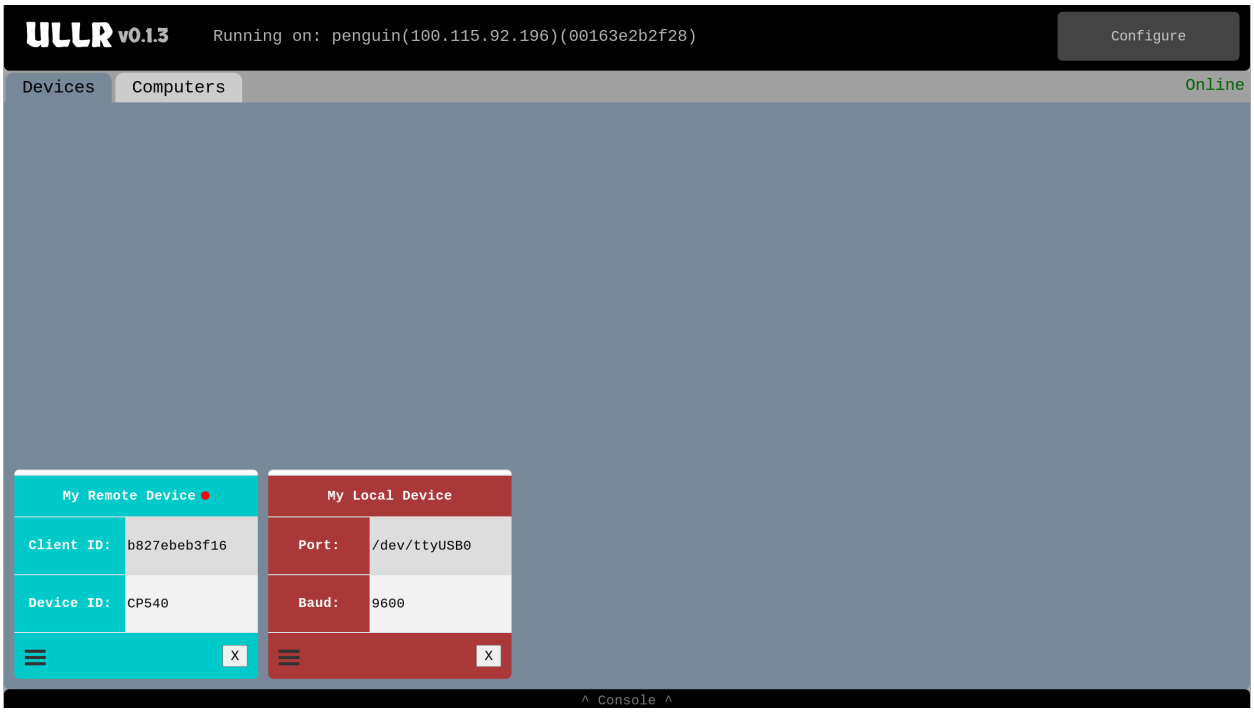

Fig. 12: The "Devices" tab showing a remote device and a local device.

Using this configuration, any message from "My Local Device" or "My Remote Device" will be sent to "My Target Software". Similarly, any message from "My Target Software" will be sent to both "My Local Device" and "My Remote Device".

When a device receives a message, either from the *MQTT broker* (the "cloud") or from a physical serial port, it will appear in the white space above the device. You can think of this similar to a receipt printer, with the newest message at the bottom and the oldest message at the top. In the image below, you can see that "My Remote Device" has received three messages and "My Local Device" has received two.

4.6.2 Using the Console

The console can be viewed by clicking the "Console" tab on the bottom middle of the screen. This provides a verbose output from the software. If you are having trouble, it is the first place to look for error feedback.

Fig. 13: The "Computers" tab showing a single local software instance.

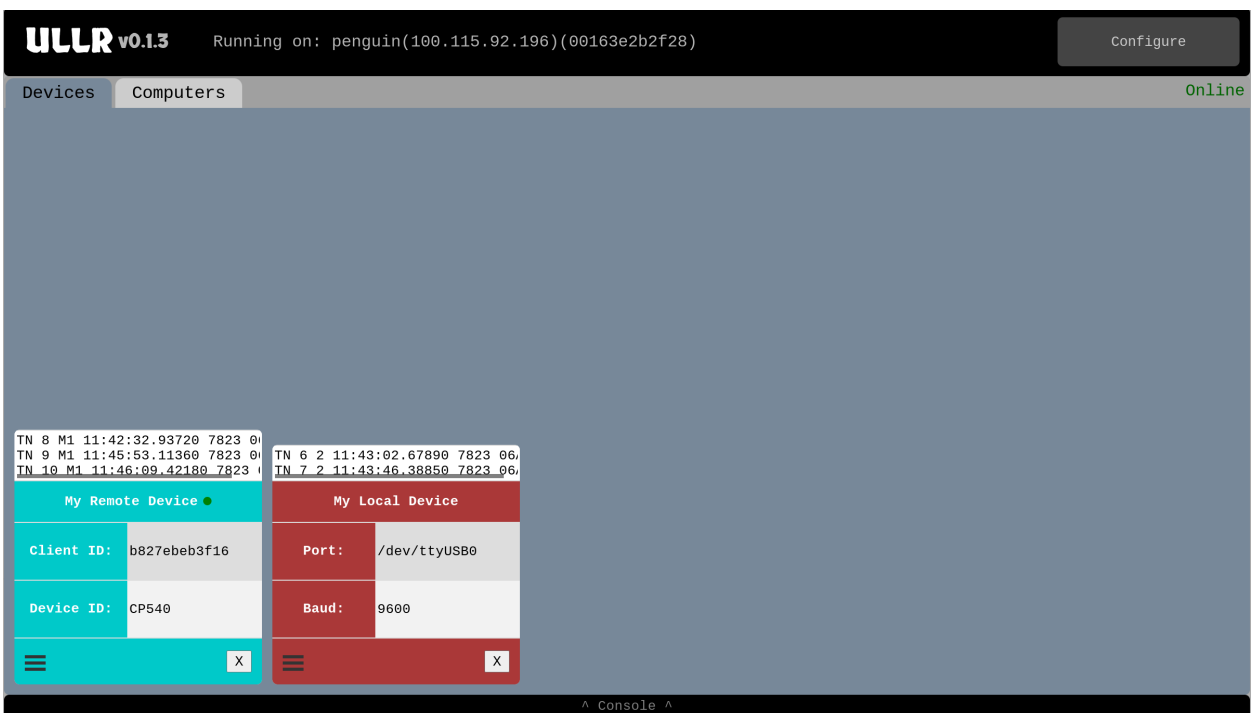

Fig. 14: The device window with several messages received.

Fig. 15: The console tab.

4.6.3 Using the Log File

Sometimes it is necessary to debug with more detail than the console provides. Ullr keeps a log file with detailed operation information. By default this file is located in the user's home directory. For example, on Windows it might be in C:\users\zhenry\.log\ullr\ullr.log. If run as superuser on Linux it will be located at /var/log/ullr/ullr.log.

The log file can also be downloaded directly through the web interface. Just open the "Configure" menu and click "Get Log File".

4.6.4 The Advanced Menu

Clicking the "hamburger" icon in the bottom left corner of a device will bring up the advanced menu for that device. There are subtle differences between the menu for local devices and the menu for remote devices, but the basic concepts are the same.

Message Transmission Settings

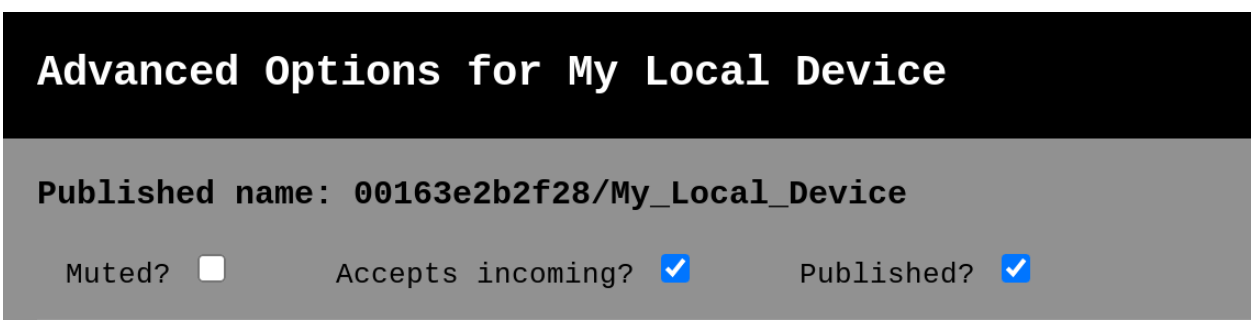

Fig. 16: The transmission settings section of the advanced menu.

The first section of the advanced menu deals with how messages are sent and received. Ticking the "Mute" checkbox will ignore any messages originating from the device. Ticking the "Accept Incoming" checkbox will allow messages to be sent to the device. And, if it's a local device, ticking the "Publish" checkbox will connect the device to the MQTT broker using the "Published name" field as the topic name.

Device Translation

The next section has to do with device translation settings. Device translation is currently only applicable to sports timing uses.

"Translating" a device means converting its message format to that of a different device. For example, if a Tag Heuer CP545 receives a new impulse on its first channel at 11:15:33.33261, it will send a message similar to the following:

```
TN       253  1 11:15:33.33261  7803       068E
```

If we were to translate the same information to Alge S4 format, it would look like this:

```
0253 SZ  11:15:33.332
```

There are a few important things to notice. While the first channel of a CP545 is 1, the first channel of an S4 is SZ. Also, take a look at the Time-of-Day: an Alge S4 has a max precision of 1/1,000th of a second, while a CP545 is precise to the 1/100,000th of a second. When we translated we lost 2 digits of precision. For this reason it is important to always translate from lower precision devices to higher precision devices.

Why translate at all? Ullr supports the connection of multiple timing devices to a single serial port, but the target software will be expecting all messages to be in a uniform format. Translation allows the connection of multiple types of devices to the same target software.

For example, with translation it is possible to have a CP540 connected to the start wand, an Alge Timy to a split, and an S4 to the photocells at the finish. All three timers can then be connected to an application that only supports the connection of one timer, such as Split Second Ski Club.

To further support this feature, you can shift the channel numbers while translating. For example, the only ports accessible on an Alge Timy, without accessories, are c0 and c1, even though it is possible to map channels c0 through c7 in Split Second. Without channel shifting we would be unable to use more than two Timys wihout ending up with overlapping channel numbers. By shifting channels we can use both built-in ports on up to 4 Timys without conflict.

For more information on setting up device translation for skiing or other sports timing, see *Advanced Ski Racing*.

Fig. 17: The translation settings in the advanced menu.

By default translation is off. To turn it on, select "True" from the dropdown menu. Next, set the source and destination settings according to your needs. The list of supported devices is growing, and *contributions* are always welcome! Finally, you can choose to shift the channels. This number can be either positive or negative as long as the resulting channel falls in the allowable range for the destination device. For example, 0-7 for an Alge Timy. It is possible to shift channels without translating to a different device format. Just select the same device for both source and destination.

Handling Late Messages

The late message feature is specific to remote devices only.

Ullr is designed to be used in portable, outdoor situations. If the quality of the internet connection is poor, messages can arrive later than expected. Depending on the use case, this can cause trouble on the receiving end. For example, Split Second software does not behave well when a competitor's start impulse arrives after their finish impulse, or when start impulses arrive out of order.

To prevent this, an on-time arrival window can be set. This is set when adding the device, and can also be edited in the advanced menu. The on-time arrival setting is the number of seconds a message can spend in transit and still be accepted by the software.

If set to 0s, all messages will be accepted regardless of transit time. If set higher than 0, any message with a longer transit time will NOT be processed and sent to other devices. It will end up in the "Late Messages" section of the advanced menu, where it can then be manually sent, copied or discarded.

Fig. 18: The late messages window showing four late messages.

To accept and send these messages, select one or more and then hit the "Send Selected" button.

When a device has received late messages, a red badge will appear above the hamburger icon with the number of late messages.

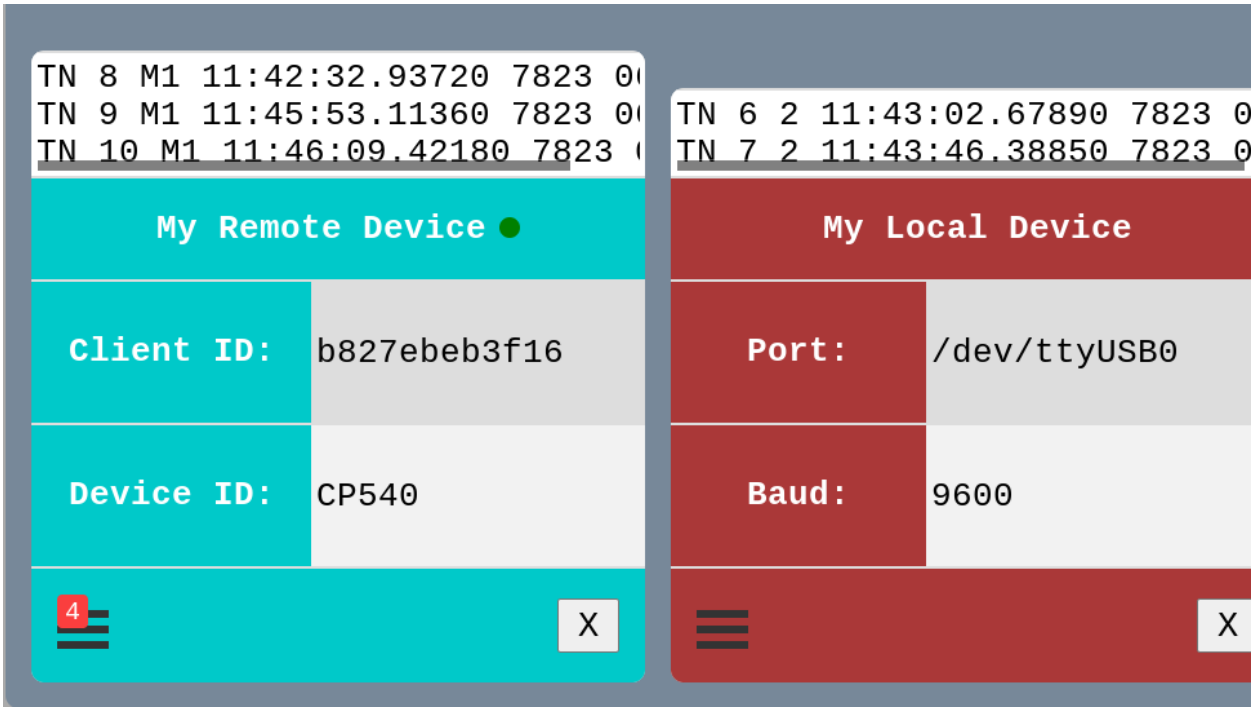

Fig. 19: Remote device showing 4 late messages.

4.7 Wireless ski race timing

This is the specific use case for which this software was originally designed. Ullr allows for FIS-legal timing of Level 3 races or lower anywhere there is an internet connection. We'll examine the setup for a homologated race step by step.

4.7.1 At the start

Wiring

The start block is connected by copper cable to two synchronized Time-of-Day timing devices, one for System A and one for System B. The System A timer is connected by serial cable to a computer running Ullr. This can be ANY computer capable of running Python applications and connecting to the internet. For example, the Raspberry Pi Zero W costs about $30 with a case and is around the size of a pocket knife. For the purposes of this tutorial we will assume that the start computer is a Raspberry Pi Zero W.

Ullr Configuration on the Raspberry Pi

Next we need to configure Ullr on the Raspberry Pi. This is fairly simple, as we only have one device to configure! Navigate to the Raspberry Pi web interface, either from the Pi itself or from another computer on the same LAN. See *Running on a Raspberry Pi* for more information on how to do this.

First we need to make sure the MQTT broker is set up correctly. See *MQTT Broker Settings* for more info on how to do this. Next, we need to set Ullr to connect to our System A timing device. Since this device is plugged directly into the Pi with a serial cable, it is a local device. Click the "Add Local Device" button in the "Configure" menu to pull up the add device dialog.

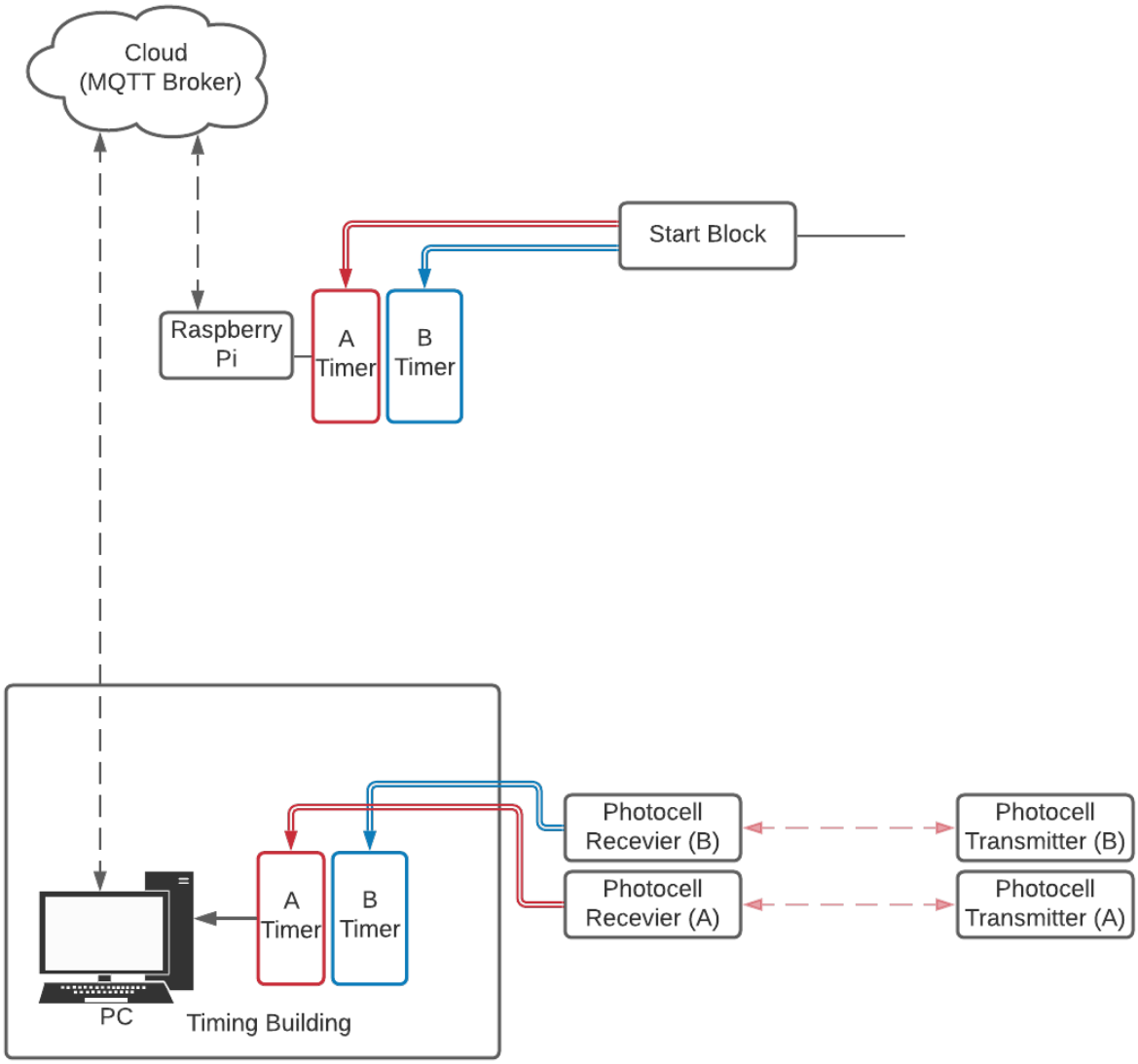

Fig. 20: Wiring and signal flow for homologated ski timing.

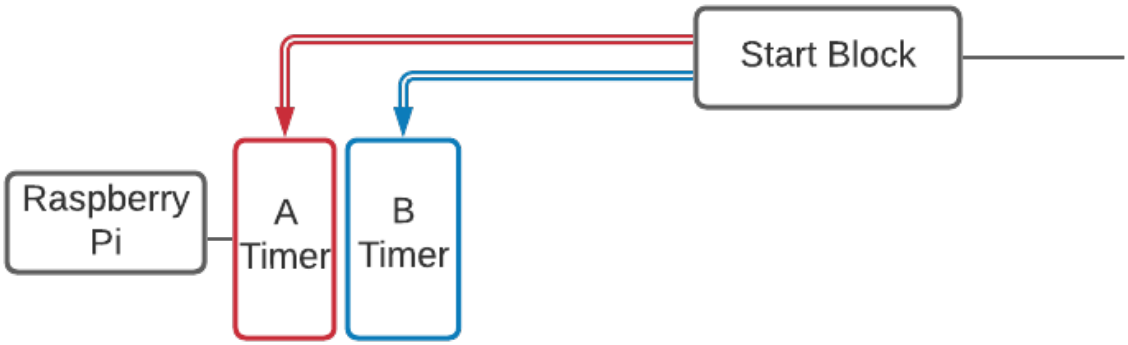

Fig. 21: Wiring at the start.

Give the device a descriptive name. For this example we'll use a TAG CP540 as our timing device, so let's name the device "CP540". The next step is to select the port. We're using a USB-serial converter cable in this example, so our port will be "/dev/ttyUSB0". If you are using a serial connection directly to the Raspberry Pi your port name may be different. The CP540 runs at 9600 baud, so we'll select that from the dropdown menu.

We'll leave the mute checkbox unticked, since we are expecting to receive messages from this device. We'll untick the "Accept Incoming" checkbox, since we don't expect the timer to receive any messages (in fact, this is against FIS rules). This also reduces some of the processing overhead. We'll leave the "Publish?" checkbox ticked as we are planning to access this device remotely.

When you're finished, the settings should look like this:

That's all the configuration needed on the Raspberry Pi! When you're done, make sure the CP540 device appears in the device window as shown below.

If it's not there, check the console for error messages and try again. Make sure to go to the "Configure" menu and save changes once you're done. Finally, click the hamburger icon in the bottom left of the device to bring up the advanced menu. Write down the info for "Published name: ". This is what we'll need to connect to this device from the finish.

Manual configuration

It's also always possible (and possibly quicker) to edit the config file directly rather than using the interactive web configuration. When running as superuser (and we should be!) on the Raspberr Pi, the config file will be located at /etc/ullr/config.ini. We can achieve the same configuration by adding the following section to the file:

```
[CP540]
type = DCE
location = local
port = /dev/ttyUSB0
baud = 9600
published = True
mute = False
accepts_incoming = False
```

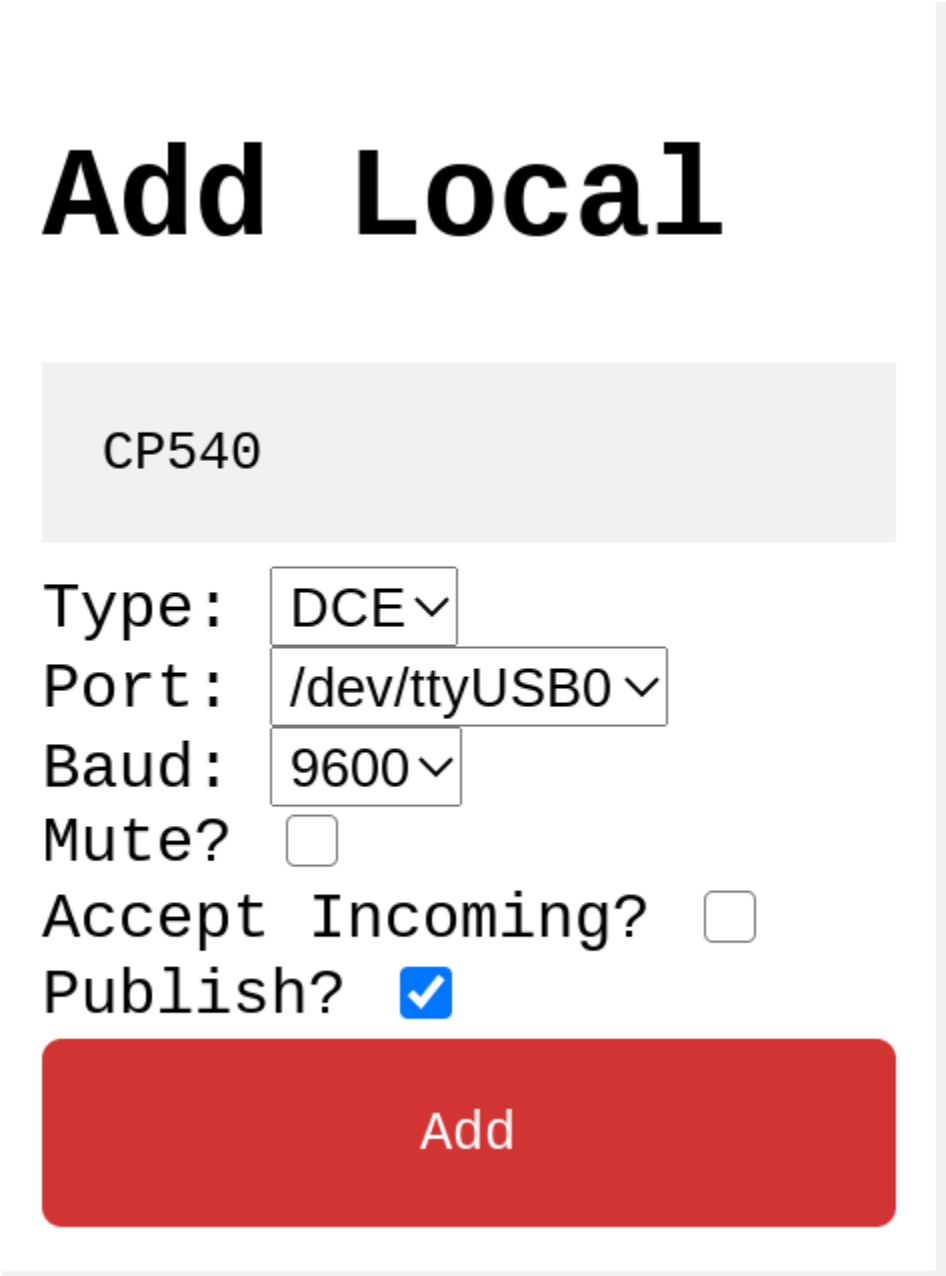

Fig. 22: Settings to add the CP540 on the Raspberry Pi.

Fig. 23: Device window with the 540 added.

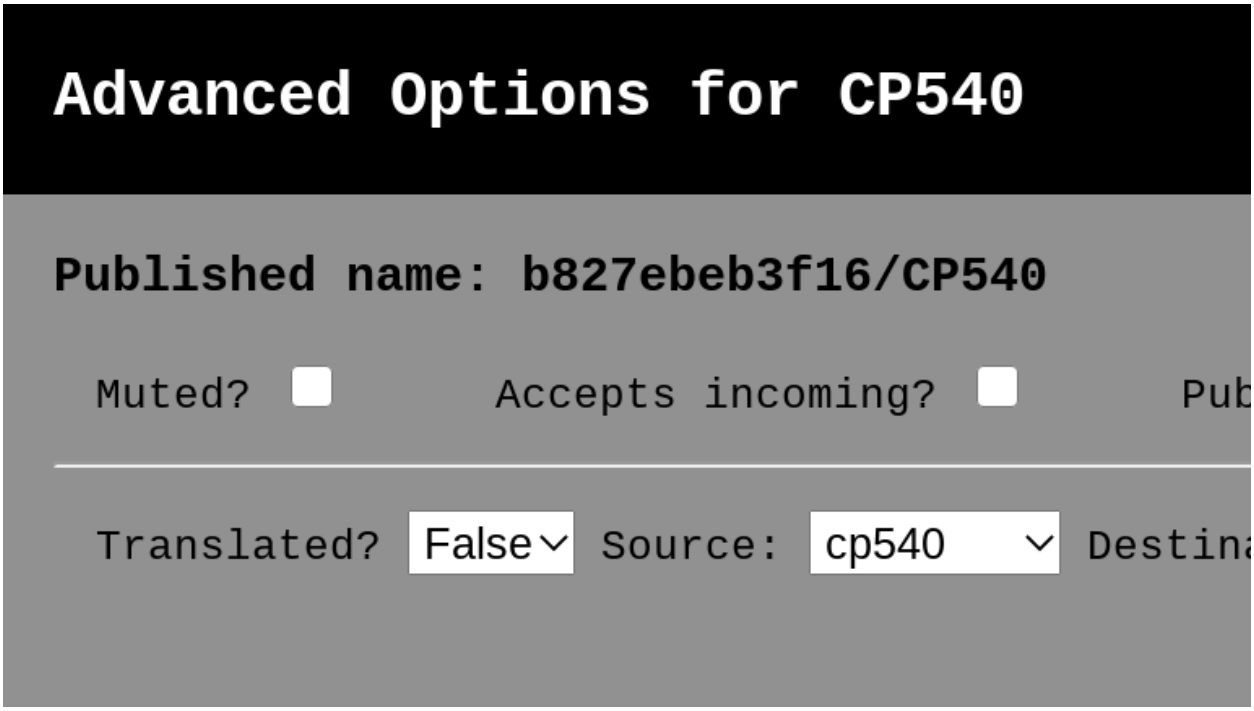

Fig. 24: CP540 advanced menu showing published name.

4.7.2 At the finish

Wiring

Wire the finish according to standard FIS rules. For example:

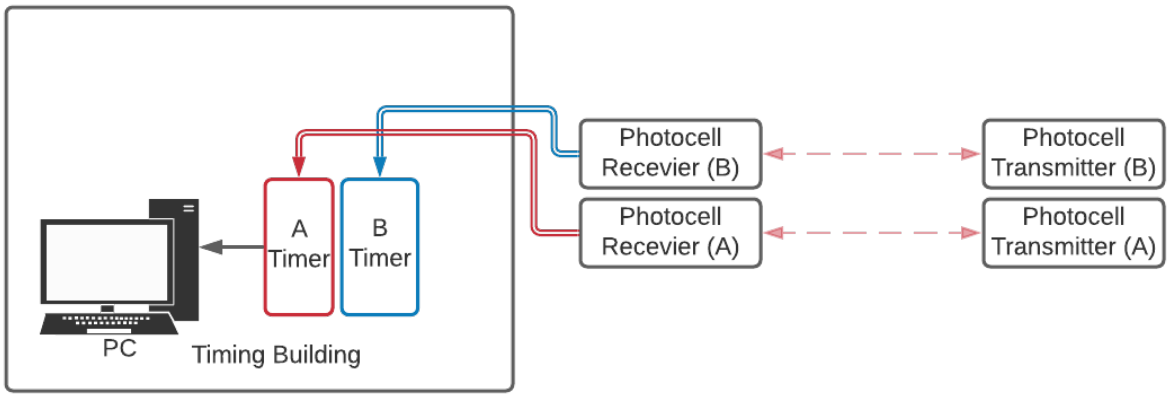

Fig. 25: A FIS legal finish setup.

All that's need to connect to the start is an internet connection and an Ullr installation. When we're done, our software will be set up like this:

Null modem setup

For Ullr to be able to connect to Split Second (or any other timing software), we'll need to setup a *virtual null modem*. You can think of a null modem as two serial ports connected by a serial cable: Ullr will connect to one port, and split second to the other. A virtual null modem is just a software implementation of this. It is the pipe that carries information from Ullr to Split Second.

There are several virtual serial port software to choose from, but for Windows the com0com project is stable and completely free and open source. A signed installer is available from the Alge website here: com0com (signed).

Once com0com is installed, we'll need to run the configuration to add a linked pair of com ports. You can choose any two port numbers you like, as long as they're not already in use. I like to use COM50 and COM51.

Ullr configuration on the finish PC

Next we'll need to get Ullr setup on the finish PC. We have two devices to add this time: our start timer (a remote DCE device), and Split Second (a local DTE device).

First, navigate to the web interface (localhost:5000) and open the "Configure" menu. The first step is to set the MQTT broker settings. They need to be the same settings as the Raspberry Pi at the start!

Next we'll add our remote start timer, using the information from the previous step. Click the "Add Remote Device" button to bring up the dialog. The device name can be anything that makes sense to you. For this guide we'll use "Start Timer". The Host ID/Device ID field is where we'll put the "published name" from the previous section. In our case it's "b827ebeb3f16/CP540".

The next field determines how late messages will be handled. If there is an interruption in internet connection, it's possible that messages from the start will arrive late. If they arrive too late it will cause unexpected behavior in Split

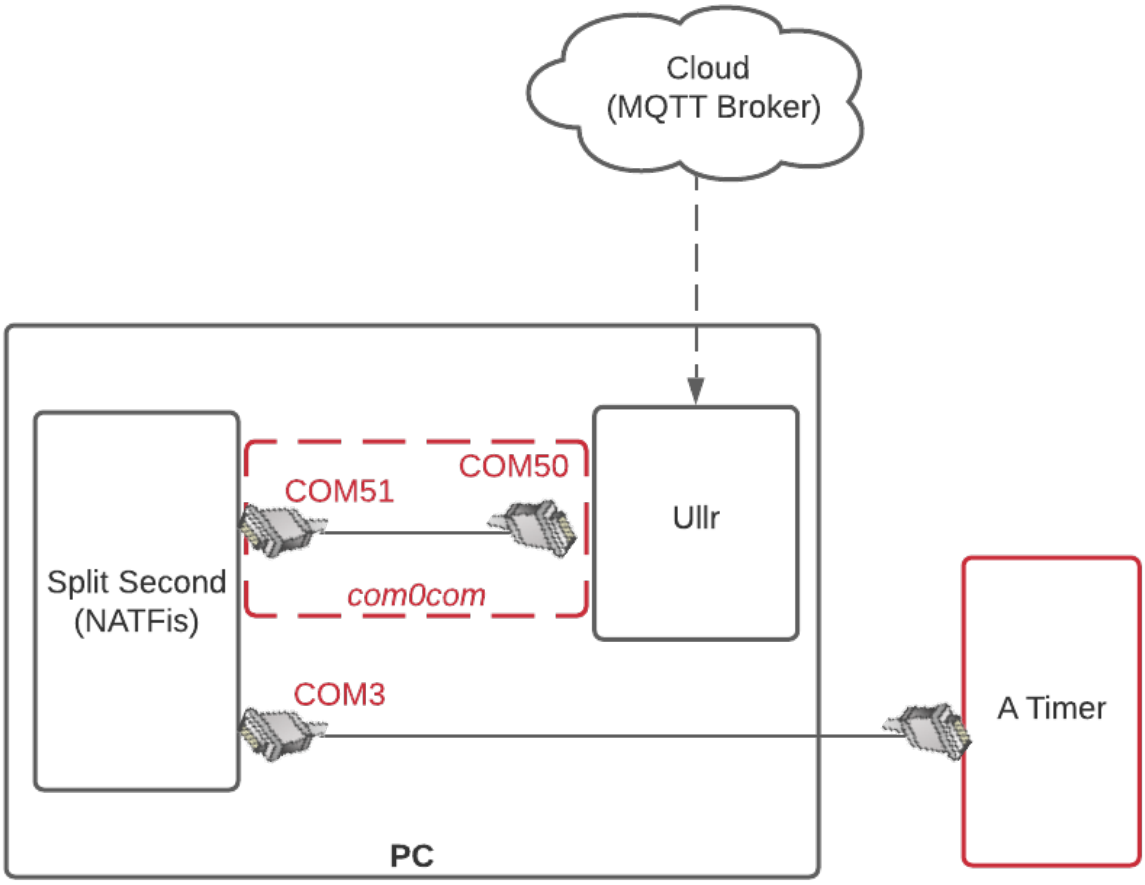

Fig. 26: Signal flow within the finish PC.

Second. For example, a start impulse that arrives after a finish impulse or a start impulse that arrives out of order will both result in trouble. Let's imagine we are running a GS on a 30s interval. An on-time arrival setting of 25s should be safe. Any message that arrives after 25s will appear in the late messages list and can manually be sent to Split Second later on. For more info on late messages, see *Handling Late Messages*.

Since our start timer is a DCE device, select DCE from the dropdown.

The next two fields are already familiar to us, and we should use the same settings we used on the Raspberry Pi at the start. Both "Mute" and "Accepts Incoming?" should be unticked.

When you're done the settings should look like this. Click the Add button.

Next we need to add Split Second (or a different timing software). Click the "Add Local Device" button.

Since we're using Split Second software in this example, let's name the device "Split Second". Since it's a piece of software, its a DTE device. Select DTE from the dropdown.

For the port, choose one of the ports in your virtual null modem pair. In our example we paired ports COM50 and COM51, so lets choose COM50. We're working with a CP540, so we'll again choose 9600 baud.

Now we have our familiar checkboxes. Since Split Second isn't sending any messages to the timer (and FIS doesn't allow it anyways), we can go ahead and tick the "Mute" box. We'll be sure to leave "Accepts incoming?" ticked as our entire goal is to send Split Second messages. We'll go ahead an untick the "Publish?" box as there is no need to access this device directly from the cloud.

When you're done, the settings should look like this. Click the Add button.

FIGURE here

We should now have two devices configured and visible in the device window, one under each tab. If you ran into any trouble check the console for error messages and try again.

At this point it's a good idea to send some test impulses from the CP540. They should appear on the virtual "timing tape" above the "Start Timer" device.

Manual configuration

Again, it's possible to add these settings to the config file directly rather than using the web interface. On Windows the config file will be located in a subdirectory of your home folder. For example, my config file is located at C:\users\zhenry\.config\ullr\config.ini. The same configuration as above can be achieved by adding the following sections:

```
[Start Timer]
type = DCE
location = remote
topic_name = b827ebeb3f16/CP540
on_time_max = 25
mute = False
accepts_incoming = False

[Split Second]
type = DTE
location = local
port = COM50
baud = 9600
published = False
mute = True
accepts_incoming = True
```

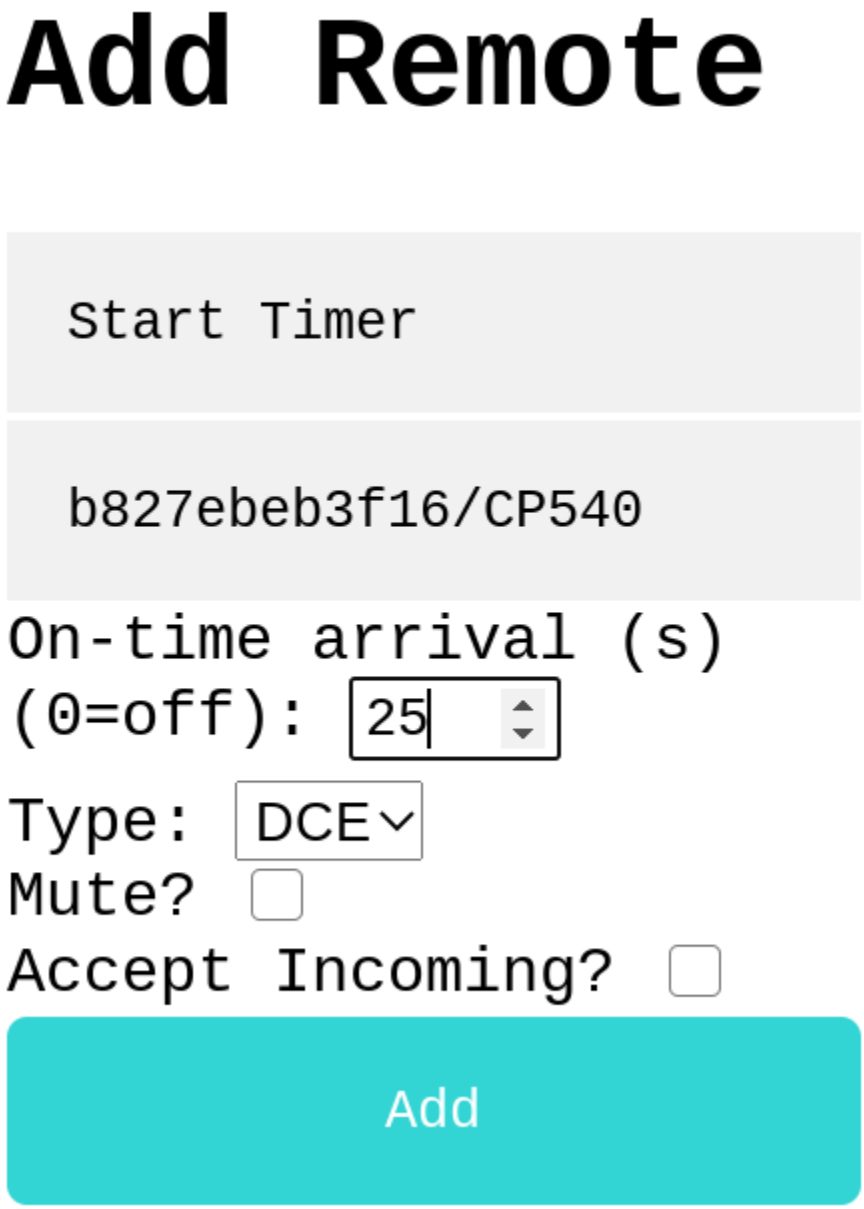

Fig. 27: Settings to add our remote start timer.

Split Second configuration

All that's left to do is configure Split Second. This is similar to the usual Split Second configuration, but this time we have two timers: our finish timer that's wired to the timing computer with a serial cable, and our start timer that is connected by Ullr and a virtual null modem.

Configure the hardwired finish timer the way you usually do.

Then, go to the second timer tab and configure the remote start timer. Choose the device name and baudrate as usual. For the port, select the other end of the virtual null modem. In our case, our null modem connects ports COM50 and COM51. We connected Ullr to COM50, so we'll connect Split Second to COM51.

Note that this only works with timing programs that support multiple timers, such as Split Second's National/FIS and Vola. Using Ski Club or another program that only supports one timer? No problem! Read on to *the next section*.

4.8 Advanced Ski Racing

What if we're using a program like Split Second's Ski Club that only supports connecting one timing device? What if we want to use 3 or more timing devices to add wireless split times? What if our devices are not all the same? Ullr can make it happen.

This section expands on the previous section, *Wireless ski race timing*. We'll start with the same basic signal flow, and add to it where necessary.

4.8.1 Connecting multiple timers to Ski Club

In our previous section we setup Ullr for wireless ski race timing. But there's one potential problem with our setup: it depends on connecting two timing devices to our timing software. NATFIs supports this, but not all timing software does. In this section we'll setup wireless timing in Ski Club, which only supports one timing device.

The wiring at the start and finish remains the same as in *the previous section*, as does the configuration of the Raspberry Pi and Ullr at the start. We just need to make some software changes in the finish.

Ullr configuration for Ski Club

We'll use the same *Null modem setup* as in our previous example.

All that changes is the flow of serial information inside the computer.

What changed from our previous setup? Instead of connecting the System A finish timer directly to the timing software, we connect it to Ullr. Ullr will intercept both the start impulses from the MQTT broker (the cloud), and the finish impulses from the physical serial port. It will then pass all of these impulses on to our null modem, which is connected to Ski Club. As far as Ski Club is concerned, it is connected to one timing device.

We need to update our Ullr configuration to reflect this change. All we need to do is add a local DCE device representing our physical timer, in addition to the remote device we added in the previous step for our start timer.

Open the Configure menu and click Add Local Device. Let's call our device "Finish Timer", and say it's connected to COM3 at 9600 baud. It should be unmuted and set to not accept incoming messages. There is no need to publish this device.

Click the Add button. You should now have two devices in the Devices tab: the remote start timer, and the local finish timer. You should also have Split Second under the Computers tab from the previous section.

You'll end up with a config file similar to this:

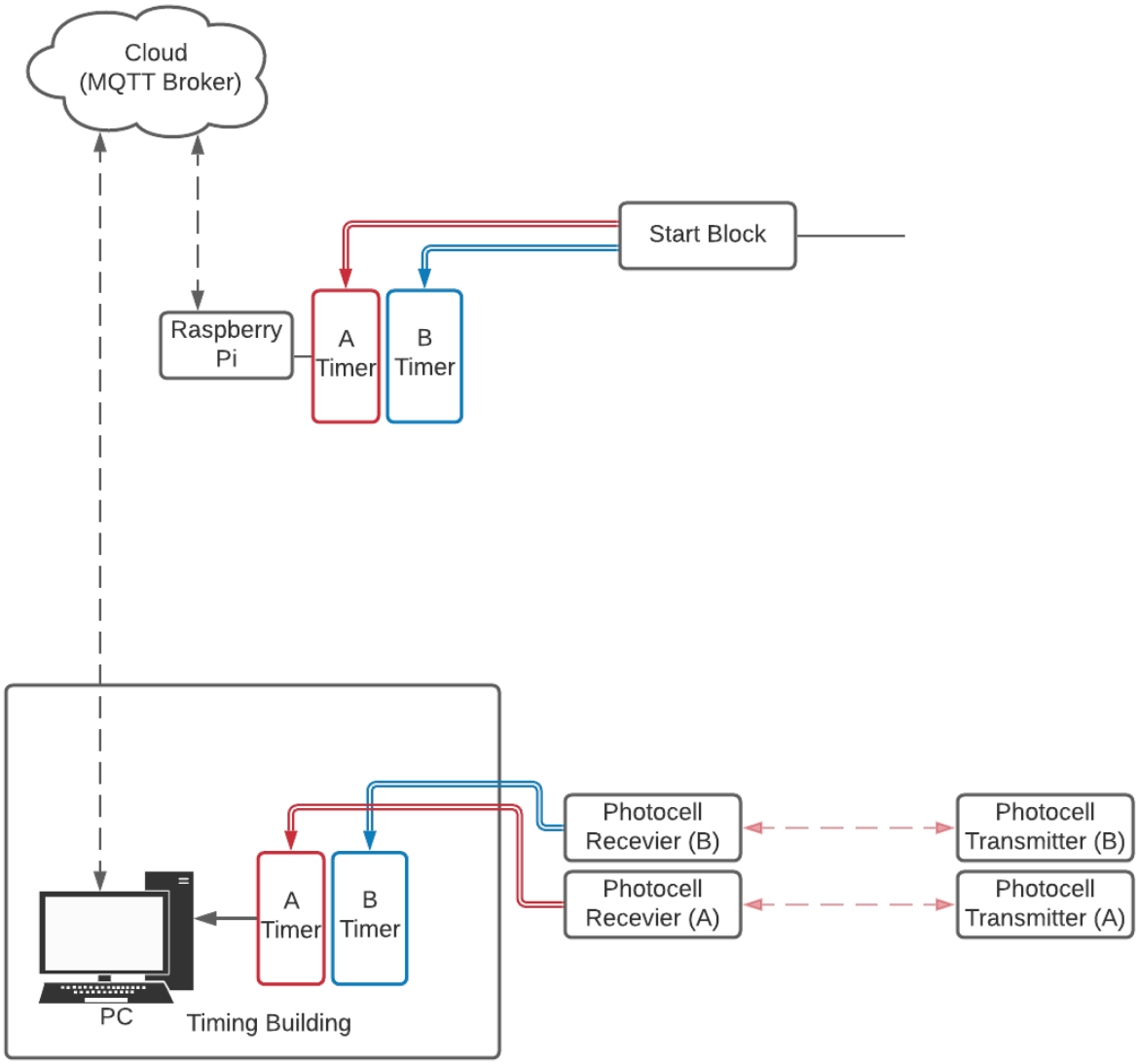

Fig. 28: Wiring and signal flow for homologated ski timing.

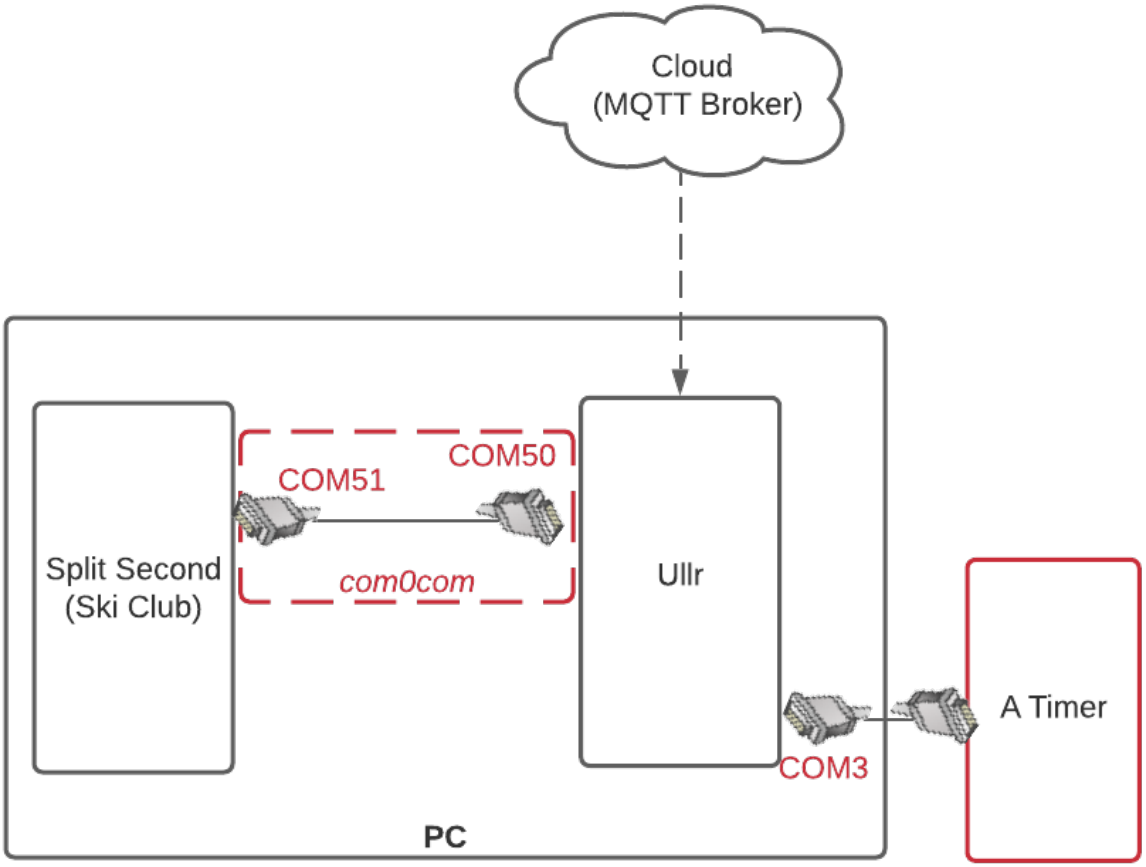

Fig. 29: Signal flow within the finish PC.

```
[Start Timer]
type = DCE
location = remote
topic_name = b827ebeb3f16/CP540
on_time_max = 25
accepts_incoming = False

[Finish Timer]
type = DCE
location = local
port = COM3
baud = 9600
published = False
accepts_incoming = False

[Split Second]
type = DTE
location = local
port = COM50
baud = 9600
published = False
mute = True
```

There are a couple things to consider with this setup. Since Ski Club doesn't know that it is reading from two timing devices, not one, it is up to you to setup input mapping in a safe and logical way. It is important not to allow channel numbers to conflict or overlap. Ski Club has no way of differentiating between an impulse on channel 1 of the remote start timer or an impulse on channel 1 of the physically connected finish timer. It may be necessary to shift channels, which will be described below.

4.8.2 Adding additional timers

There may be times when it's convenient to connect even more than 2 timers to our timing software. Consider the setup with a wireless speed trap below:

We've kept our homologated wireless setup at the start and our hardwired finish timer, but added a third timer to handle a speed trap, for this example a Microgate Racetime2. Since there are no homologation requirements for splits and speed traps it's possible to use a radio transmitted photocell such as the Microgate.

However, even Vola and NATFis software only allow the connection of 2 timers, and now we have three. We can set Ullr up to send all of the timing information over one port, as below.

It's just a matter of *configuring* Ullr correctly. For this example, we have one remote device, our start timer. We have two local devices connected to the timing PC with a serial cable: the finish timer and the split timer. Ullr will intercept messages from all 3 devices and pass them on to Split Second through our null modem.

If we configure Ullr through the Web UI, our device window will look something like this:

FIGURE here

We could also just manually enter the info into the *config file*. We'll end up with something like this:

```
[Start Timer]
type = DCE
location = remote
topic_name = b827ebeb3f16/CP540
```

(continues on next page)

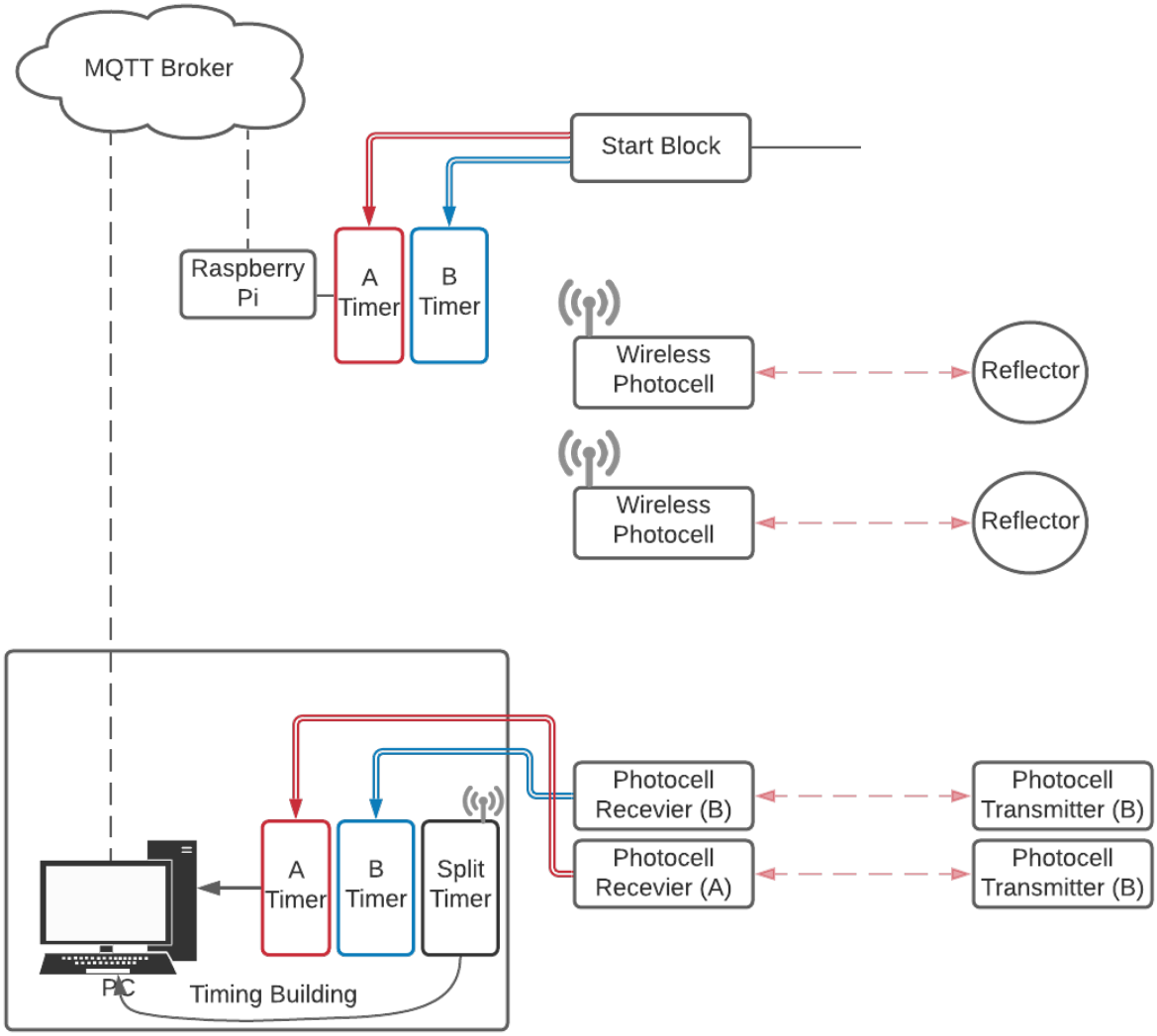

Fig. 30: Signal flow with a homologated wireless start and non-homologated speed trap.

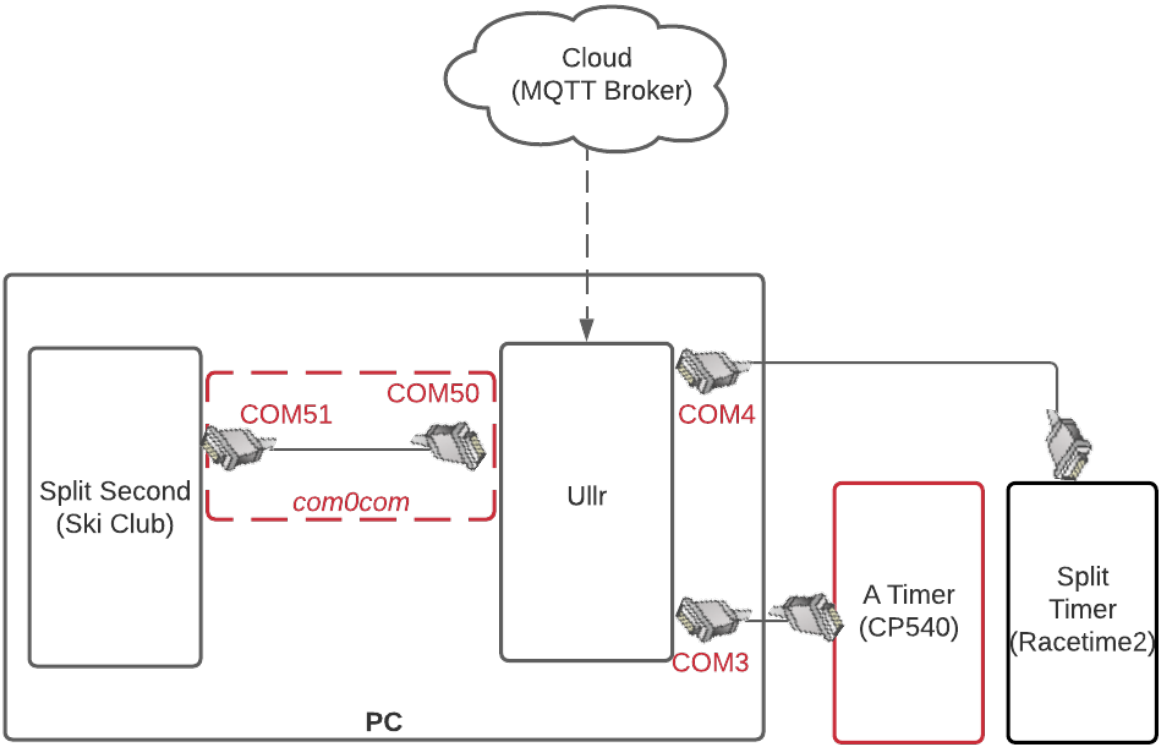

Fig. 31: Signal flow within the finish PC with multiple timers.

(continued from previous page)

```
on_time_max = 25
accepts_incoming = False

[Finish Timer]
type = DCE
location = local
port = COM3
baud = 9600
published = False
accepts_incoming = False

[Split Timer]
type = DCE
location = local
port = COM4
baud = 9600
published = False
accepts_incoming = False

[Split Second]
type = DTE
location = local
port = COM50
baud = 9600
published = False
mute = True
```

This example assumes that all three of your timers (start, finish and split) are the same. Since we are sending information from all three to Split Second over a single serial port, Split Second will assume that all of the messages are in the same format. Mixed format messages will cause errors.

What if all 3 timers aren't the same? How can we get the messages into a uniform format that Split Second can understand? That's where translation comes into play.

4.8.3 Translation Settings

4.9 Connecting to a Remote Display Board

4.10 Command Line Operation

Ullr can take several useful command line arguments, which will be described below. Ullr can also be run interactively through the terminal, without the use of the web interface. This can be useful for reducing processing overhead, or for monitoring operation on headless installations.

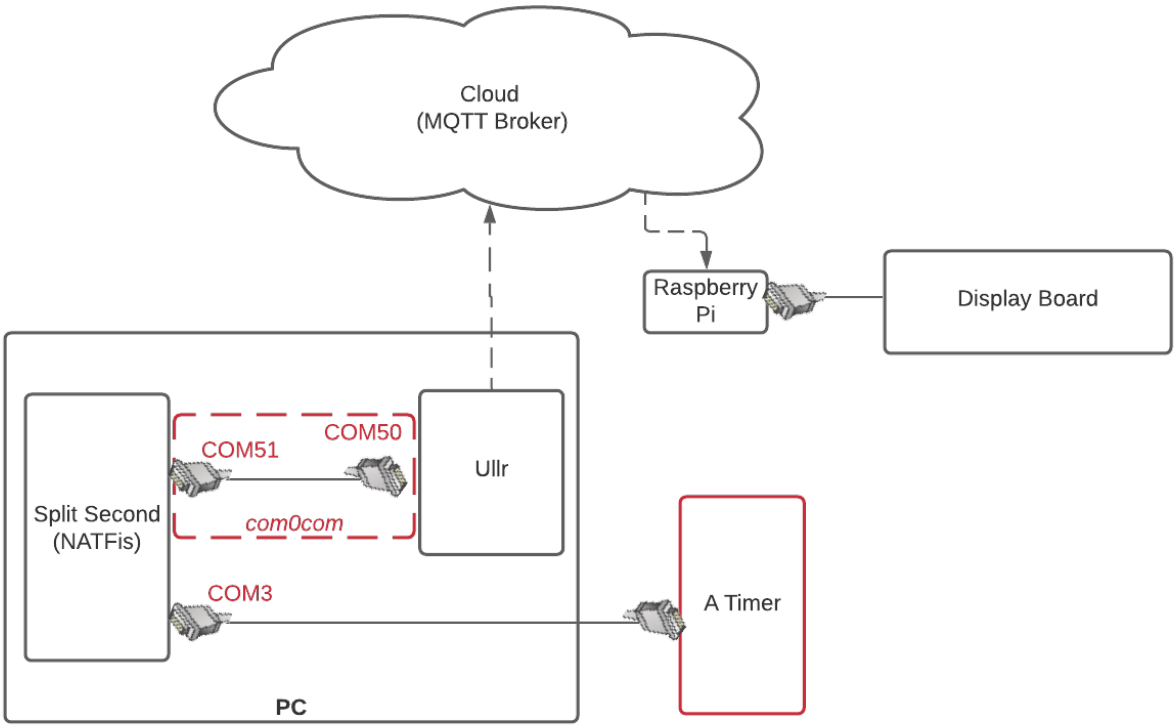

Fig. 32: Signal flow for a wireless display board.

4.10.1 Command Line Arguments

Ullr accepts several useful command line arguments.

Ignore the config file

```
ullr --empty
```

This starts Ullr with a blank, default configuration regardless of any settings in the configuration file. This is useful for fixing broken config files or for starting fresh.

Use an arbitrary config file

```
ullr --file my-config.ini
```

This loads devices from a user-specified config file in an arbitrary location rather than from the standard config file.

Specify WebUI port

```
ullr --uiport 1234
```

By default, Ullr serves the Web UI on port 5000 of the local machine. You can use the –uiport flag to specify a different port. This is useful if port 5000 is not available, or if multiple instances of Ullr are to be run on the same machine (be careful doing this!)

Open WebUI on run

```
ullr --popup
```

This opens a browser to the web interface when Ullr is run. This is the defualt for Windows .msi package installations.

4.10.2 Running in the Terminal

```
ullr --nowebui
```

This runs Ullr in the terminal without starting the web interface. Configuration can be done interactively, though not all of the WebUI features are available, most notably the late message handling features.

4.11 Using the Config File

Ullr can be configured using the *interactive web interface*. However, configuration can be achieved by simply modifying the config file.

The Ullr config file follows INI format. Each section represents a device, with the exception of the DEFAULT and $mqtt sections.

4.11.1 Finding the config file

A default config file is created the first time Ullr is run. Before that, the config file will not exist.

The location of the config file depends on your operating system, and whether you are running as *superuser (root)* or as a regular user.

On Windows, the config file will be located in the .config subfolder of your home directory. For example, C:\users\zhenry\.config\ullr\config.ini.

The same applies for Linux if run as a regular user. For example, /home/zhenry/.config/ullr/config.ini.

If run on Linux as superuser the config file will be located at /etc/ullr.config.ini.

4.11.2 Default config settings

Let's take a look at the 'DEFAULT' section of the config file. These are the device settings that Ullr falls back to if you don't specify anything different.

```
[DEFAULT]
type =
location =
mute = False
accepts_incoming = True
port =
topic_name = defualt_client/default_device
baud = 9600
published = False
on_time_max = 0
translated = False
translated_from = None
translated_to = None
channel_shift = 0
mqtt_broker_url = 57edaf7763054ccc91c1c8b6e646a155.s1.eu.hivemq.cloud
mqtt_broker_port = 8883
mqtt_broker_user = PiTiming
mqtt_broker_pw = Mammoth1
```

Type and location have no default. Type can be either DCE or DTE, and location can be either remote or local. The rest of the default settings are fairly self-explanatory.

4.11.3 MQTT broker settings

The *MQTT broker* settings are under a special section named '$mqtt'. We'll stick with the default settings for this example, but you can use any broker settings you'd like. Using a private broker provides added security.

```
[$mqtt]
mqtt_broker_url = 57edaf7763054ccc91c1c8b6e646a155.s1.eu.hivemq.cloud
mqtt_broker_port = 8883
mqtt_broker_user = PiTiming
mqtt_broker_pw = Mammoth1
```

4.11.4 Adding a device

To add and configure a device, all we have to do is add a section with the device name. For example, say we want to add a local DCE device called "Finish Timer". Let's say it's 9600 baud and plugged into COM3. We'd add the following section:

```
[Finish Timer]
type = DCE
location = local
port = COM3
baud = 9600
```

That's it! Note that any key value we didn't specify will fall back to the default: this device won't be muted, it will accept incoming messages, and it won't be translated or published. To change any of these settings, we just have to add the appropriate key value to the section.

4.12 Project Links

- Ullr GitHub homepage
- Ullr Source (tarball)
- Latest Release Page

4.13 External Links

- The FIS International Competetion Rules
- HiveMQ MQTT broker
- Timing Guys forums

4.14 Contributing to Ullr

Contributions to Ullr are welcome and encouraged! The backbone of the project is written in Python, but there is a fair amount of HTML, JavaScript, and CSS required by the web interface.

Not a coder? This documentation could use some work! Adding detail and clarity, or explaining a unique use case is much appreciated.

4.14.1 Ullr is Developed on Github

I use github to host code, to track issues and feature requests, as well as accept pull requests. See the Ullr GitHub homepage.

4.14.2 Use Github Flow, So All Code Changes Happen Through Pull Requests

Pull requests are the best way to propose changes to the codebase. Pull requests are actively welcomed!

1. Fork the repo and create your branch from *main*.
2. Make your changes.
3. If your code changes how Ullr works, update the docs.
4. Make sure your code lints (works).
5. Issue that pull request!

4.14.3 Any contributions you make will be under the GNU GPL v3 license.

In short, when you submit code changes, your submissions are understood to be under the same GPL License that covers the project.

4.14.4 Report bugs using Github's issues

Ullr uses GitHub issues to track public bugs. Report a bug by opening a new issue; it's that easy!

4.14.5 References

This document was adapted from the open-source contribution guidelines for Facebook's Draft