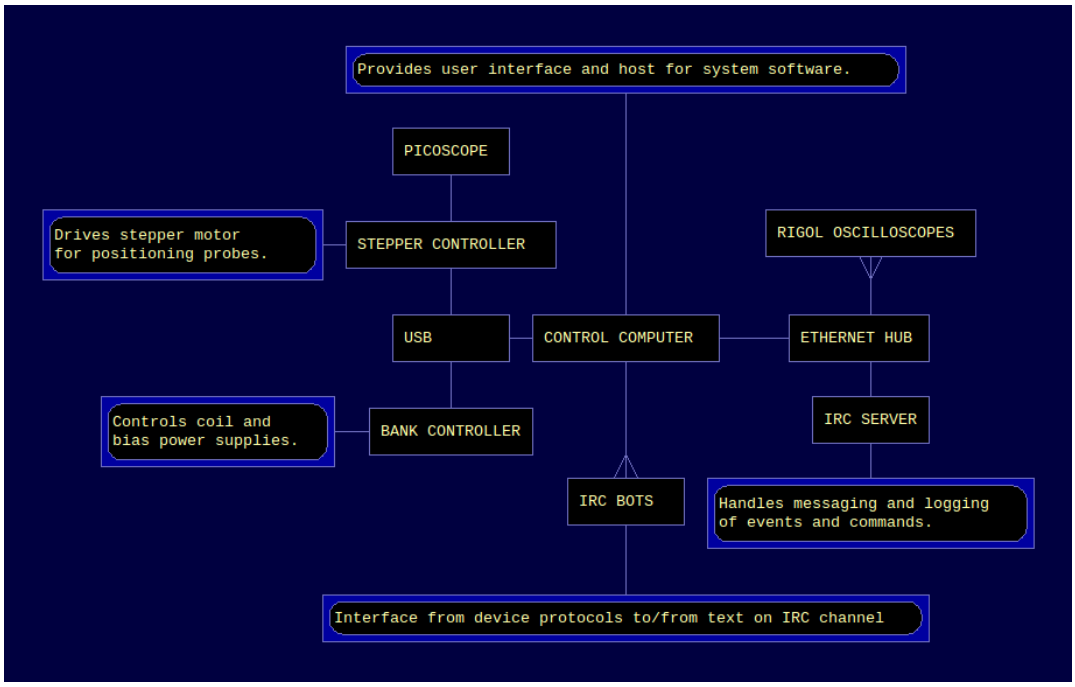

Figure 1: Overall system design

1 The Control System

This section describes the embedded control system and user interaction. The latest system software and documentation can always be found on github at `https://github.com/hedj/fusion/`.

Flowcharts in this section use the visual language from the DRAKON project. See `http://drakon-editor.sourceforge.net/language.html` for details.

1.1 The Control Computer

The control computer is a standard PC presently running Windows 10. It shares its connection to the internet via ethernet, which is connected to a hub. Oscilloscopes and the IRC server hang off this hub. At present, all the software agents described later run on the control computer, although in principle it would be possible to run many of them on other computers.

1.2 The Bank Controller

The bank controller is a Controllino Mega; an Arduino-based programmable-logic controller. The bank controller is in charge of the following functions:

- Capacitor-bank and HV pulse generation

- HV voltage control
- Capacitor bank charge control
- Audible alert before pulse

The top-level flow of the bank controller is shown in Fig. 2.

The serial protocol used is nonstandard; a definition can be found at `https://github.com/hedj/nslc`. It is designed to facilitate debugging using a serial terminal; nslc is a simple frame-based protocol using line-feeds as end-of-frame markers and whitespace for escape. It is also checksummed for reliability. The particular checksum is only 8 bits in size, and is optimised for speed of implementation.

The most important thing to emphasise is that the controller is implemented as a tight loop reacting to the change in time. There are no blocking wait statements; in this way we ensure that input can continue to be received from the front panel switches or serial as the process runs. This allows the safe abort of the system regardless of its state, which is rather important.

The two operations which consume a reasonable amount of time and are thus implemented as conditional actions in the inner loop are pulsing and charging, respectively Figs. 3 and 4.

1.3 The Stepper Controller

The stepper controller is an Arduino UNO; four pins and mosfets are used to switch an externally-supplied 12V to the poles of a stepper motor. The stepper motor accepts simple serial commands; e.g 'F5' for forward 5 milimeters, or 'R10' for 'Reverse 10 mm'.

1.4 The IRC Server

The IRC server is a raspberry pi running Linux. At present it has the fixed IP address 192.168.137.3. The IRC server runs NGIRCD, an open-source IRC daemon. All user-facing software connects to this IRC server. In this fashion, we can easily view and log all messages and commands being issued or responded to by the various subsystems.

1.5 The Agents

Each important subsystem has its own IRC bot; there is a bankbot which translates commands and messages for the bank controller, a stepperbot for the steppers, and so on. There are also some utility IRC bots. The talkbot does text-to-speech translation, and the sequencebot can perform complex sequences of tasks, including sending messages to the other bots.

The complete list of bots (Agents) presently in use is as follows:

- bankbot : controls the cap bank and HV power supplies

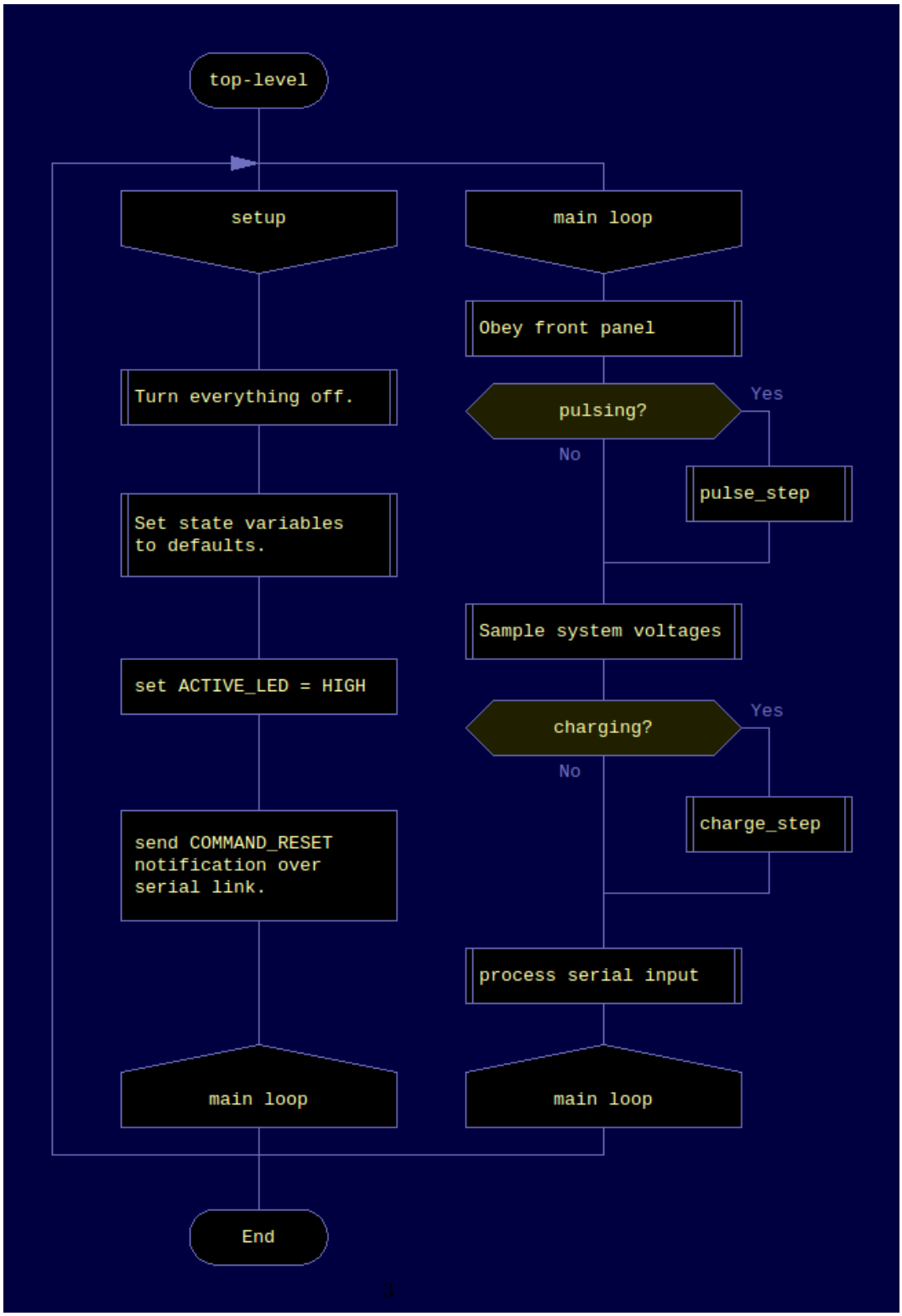

Figure 2: Top-level flow of the bank controller

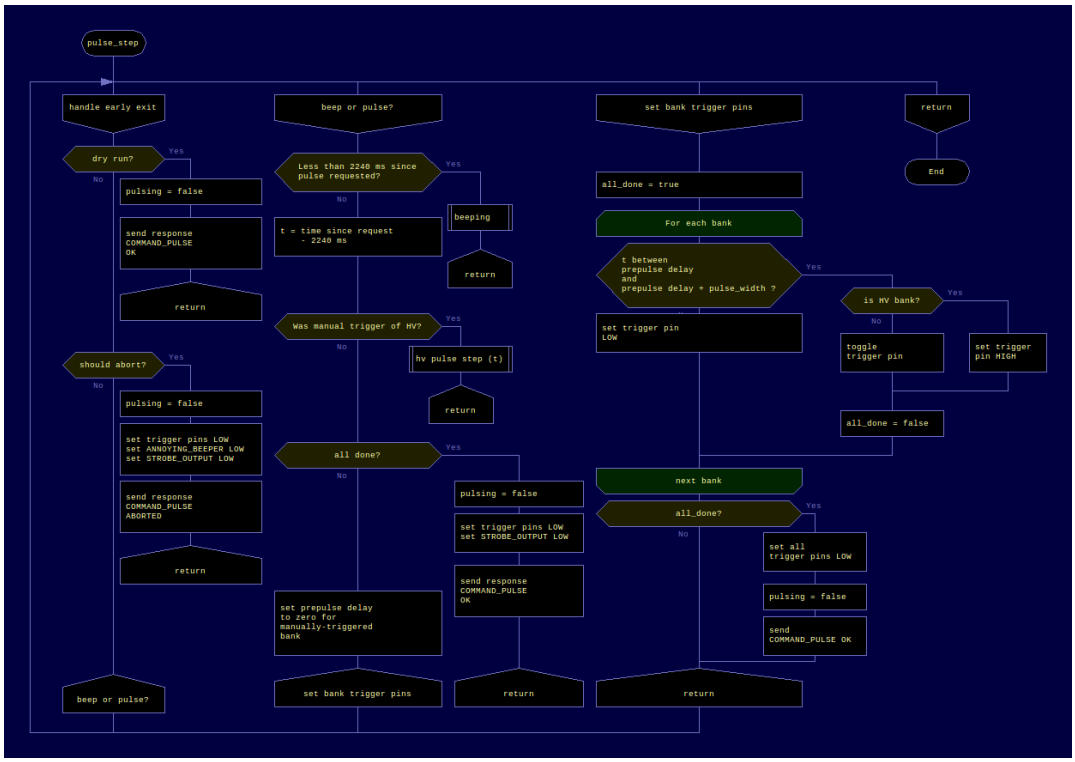

Figure 3: Pulse Step processing

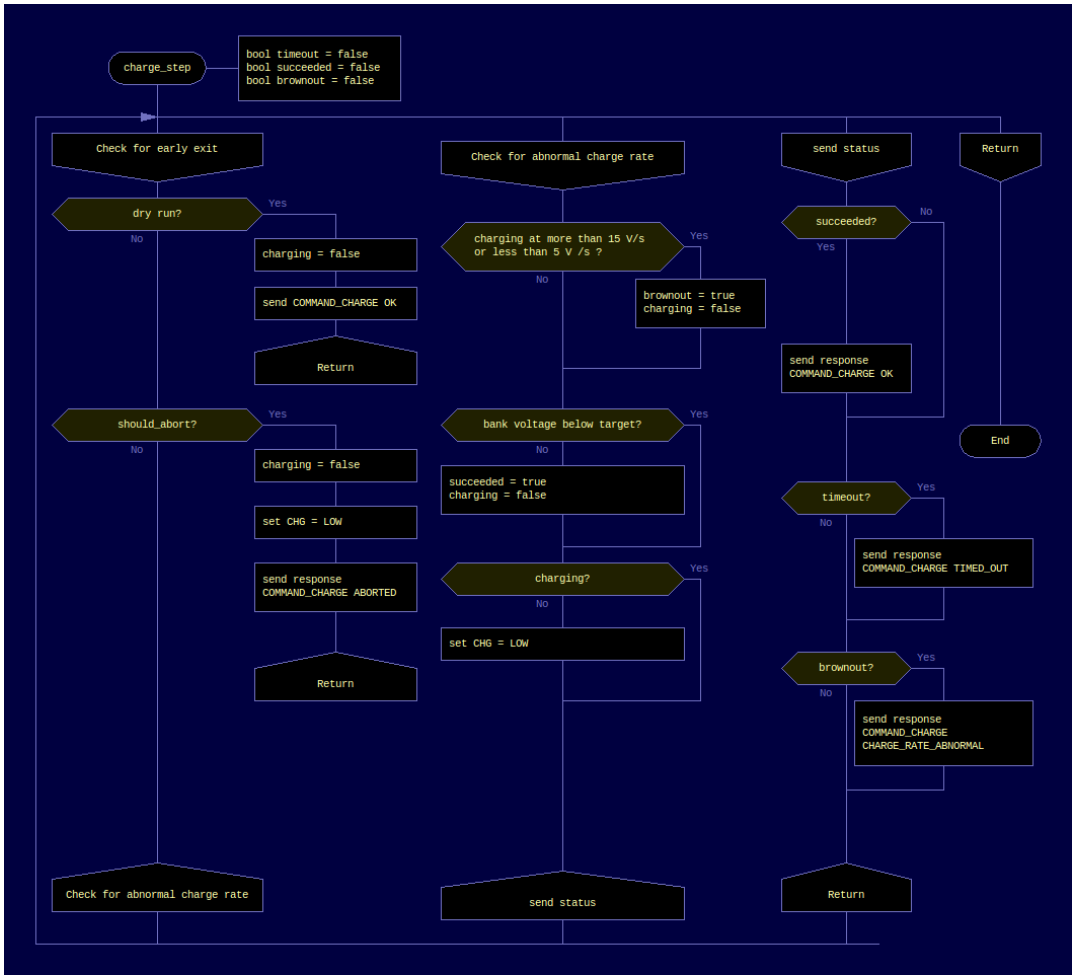

Figure 4: Charging step logic

- ipfsbot : uploads files or directories to IPFS
- picobot : controls the Picoscope 3405D oscilloscope
- rigolbot : controls the two DS10x4z oscilloscopes
- talkbot : says things out loud - e.g countdowns
- robot : runs complex sequences of commands

1.6 Using the System

The HexChat application is set up to connect to the IRC server and become the system operator; it then connects to the channel called #system. The run_bots.bat batch-file (linked to on the desktop) will invoke all of the bots, which will then initialise themselves and connect to the same channel.

Sending any of the bots a message of the form "BOTNAME: help", e.g 'bank: help', will cause that bot to reply with a description of all the commands supported by that bot. To issue a command CMD, you can just type "BOTNAME: CMD".

1.7 bankbot Command summary

The bank control robot, bankbot, understands the following commands

```
get <PARAMETER_NAME>
set <PARAMETER_NAME> <VALUE>
reset
dry_run <1|0>
pulse
\abort
```

PARAMETER_NAME can be one of the following:

```
pulse_width
pulse_delay
charge_enable
charge_power
hv_state
hv_voltage
bank_voltage
switch_state
```

The symbolic names True, False, On, and Off, may be used in place of 1 and 0.

1.8 ipfsbot Command summary

The ipfsbot understands only 'help' and 'publish'. The command 'publish' expects a path as an argument and publishes all the files under that path to the IPFS filesystem. The identifying hash value is returned as a URL enabling remote access to the published file.

1.9 picobot Command summary

The picoscope control robot, picobot, understands the following commands:

```
start_capture  :  start_capture <TIMEOUT_MS> arms the picoscope for capture
channel_config  : configures a chosen channel;
                    e.g  channel_config <NAME> <COUPLING> <VOLTAGE_RANGE>
trig_threshold    sets the voltage threshold (in Volts used for triggering
trig_channel   :  sets the channel used for triggering;  e.g "trig_channel A"
status         :  shows device configuration
reset_scope    :  disconnects and reconnects scope
write_data  :  write_data <PATH_PREFIX> writes the last data set to disk in HDF5 format
channel_names  :  'channel_names <NAME_1> <NAME_2> <NAME_3> <NAME_4>
                   names the four device channels'
help :  shows this useful message
```

1.10 rigolbot Command summary

Rigolbot only supports DS10x4z devices at present. The commandset is:

```
channel_names :    channel_names <DEVICE_NAME> <NAME_1> <NAME_2> ...
                     names the channels of the chosen scope
add_device :       add_device <IP> <NAME>  adds the device at IP and names it NAME
                     (e.g add_device 192.168.137.10 pulse_scope
remove_device :    remove_device <NAME> removes the named device
status     :       shows information about connected Rigol scopes
write_data :       write_data <PATH> writes the current screen contents of each rigol scope to <PA
help :             shows this useful message
```

1.11 talkbot Command summary

Talkbot speaks any message sent to it. If you want talkbot to say 'Hello', you just type 'say: Hello' in the #system IRC channel.

1.12 robot Command summary

The sequencing robot, simply called robot, has a complicated commandset. This is because robot.py is essentially a Python interpreter connected to the IRC channel. The initial set of commands is given in the file 'macrosystem.py', and new commands can be defined at runtime.

The best way to learn to use the robot is to enter 'robot: help' into the #system IRC channel; nevertheless, a brief summary is as follows:

The robot has a few primitives built in:

- help – shows a help message
- emit(s) – writes something to the channel, e.g emit("say: 5")
- wait(s,timeout=1) waits to hear the string s from the IRC channel, or panics after timeout
- process_line(s) interprets a string

The rules for process_line are as follows:

```
lines starting with @ are executed
lines starting with ! are emitted to the channel
lines starting with # are treated as comments (ignored)
```

New commands can be then defined by sending robot messages of the form

```
def function(arg1, arg2, ..) ->  python_code_line_1; python_code_line_2; ...
```

In 'macrosystem.py', this facility is used to build up some useful commands:

```
inputs = [
  'RUNDIR = r"c:\data\default_rundir"',
  'def setup_rigols() -> emit("rigol: add_device 192.168.137.10 rigol_1"); wait("Connected to rigol", 5)',
  'def charge_enable(s) -> emit("bank: set charge_enable " + s)',
  'def charge_power(s) -> emit("bank: set charge_power " +s)',
  'def countdown(n) -> [ (emit("say: "+str(d)), wait("OK", 5)) for d in range(n,0,-1) ]',
  'def setup() -> setup_rigols(); countdown(5); charge_power("on"); sleep(0.5); charge_enable("on")',
  'def shutdown() -> charge_power("off"); sleep(0.1); charge_enable("off"); emit("Shutdown complete")',
  'def charge(V) -> emit("bank: charge "+str(V)); wait("charge : OK, value = 0", 30)',
  'def start_capture() -> emit("picoscope: start_capture 5000")',
  'def pulse() -> emit("bank: pulse"); wait("pulse : OK, value = 0", 10)',
  'def collect_data(dir) -> emit("picoscope: write_data " + dir); emit("rigol: write_data " + dir)',
  'def shot(V, dir) -> charge(V); start_capture(); pulse(); sleep(1); collect_data(dir); sleep(10)',
  'def run_file(filename) -> state.seq = 1; [  process_line(line.strip()) for line in open(filename).readlines() ]'
]
```

In the above you can also see that environment variables can be set and used as you might expect.