Debugging

This document describes some of the Klipper debugging tools.

Running the regression tests

The main Klipper GitHub repository uses "github actions" to run a series of regression tests. It can be useful to run some of these tests locally.

The source code "whitespace check" can be run with:

```
./scripts/check_whitespace.sh
```

The Klippy regression test suite requires "data dictionaries" from many platforms. The easiest way to obtain them is to download them from github. Once the data dictionaries are downloaded, use the following to run the regression suite:

```
tar xfz klipper-dict-20??????.tar.gz
~/klippy-env/bin/python ~/klipper/scripts/test_klippy.py -d dict/
~/klipper/test/klippy/*.test
```

Manually sending commands to the micro-controller

Normally, the host klippy.py process would be used to translate gcode commands to Klipper micro-controller commands. However, it's also possible to manually send these MCU commands (functions marked with the DECL_COMMAND() macro in the Klipper source code). To do so, run:

```
~/klippy-env/bin/python ./klippy/console.py /tmp/pseudoserial
```

See the "HELP" command within the tool for more information on its functionality.

Some command-line options are available. For more information run: `~/klippy-env/bin/python ./klippy/console.py --help`

Translating gcode files to micro-controller commands

The Klippy host code can run in a batch mode to produce the low-level micro-controller commands associated with a gcode file. Inspecting these low-level commands is useful when trying to understand the actions of the low-level hardware. It can also be useful to compare the difference in micro-controller commands after a code change.

To run Klippy in this batch mode, there is a one time step necessary to generate the micro-controller "data dictionary". This is done by compiling the micro-controller code to obtain the **out/klipper.dict** file:

```
make menuconfig
make
```

Once the above is done it is possible to run Klipper in batch mode (see installation for the steps necessary to build the python virtual environment and a printer.cfg file):

```
~/klippy-env/bin/python ./klippy/klippy.py ~/printer.cfg -i test.gcode -o
test.serial -v -d out/klipper.dict
```

The above will produce a file **test.serial** with the binary serial output. This output can be translated to readable text with:

```
~/klippy-env/bin/python ./klippy/parsedump.py out/klipper.dict test.serial
> test.txt
```

The resulting file **test.txt** contains a human readable list of micro-controller commands.

The batch mode disables certain response / request commands in order to function. As a result, there will be some differences between actual commands and the above output. The generated data is useful for testing and inspection; it is not useful for sending to a real micro-controller.

Motion analysis and data logging

Klipper supports logging its internal motion history, which can be later analyzed. To use this feature, Klipper must be started with the API Server enabled.

Data logging is enabled with the `data_logger.py` tool. For example:

```
~/klipper/scripts/motan/data_logger.py /tmp/klippy_uds mylog
```

This command will connect to the Klipper API Server, subscribe to status and motion information, and log the results. Two files are generated - a compressed data file and an index file (eg, `mylog.json.gz` and `mylog.index.gz`). After starting the logging, it is possible to complete prints and other actions - the logging will continue in the background. When done logging, hit `ctrl-c` to exit from the `data_logger.py` tool.

The resulting files can be read and graphed using the `motan_graph.py` tool. To generate graphs on a Raspberry Pi, a one time step is necessary to install the "matplotlib" package:

```
sudo apt-get update
sudo apt-get install python-matplotlib
```

However, it may be more convenient to copy the data files to a desktop class machine along with the Python code in the `scripts/motan/` directory. The motion analysis scripts should run on any machine with a recent version of Python and Matplotlib installed.

Graphs can be generated with a command like the following:

```
~/klipper/scripts/motan/motan_graph.py mylog -o mygraph.png
```

One can use the `-g` option to specify the datasets to graph (it takes a Python literal containing a list of lists). For example:

```
~/klipper/scripts/motan/motan_graph.py mylog -g
'[["trapq(toolhead,velocity)"], ["trapq(toolhead,accel)"]]'
```

The list of available datasets can be found using the `-l` option - for example:

```
~/klipper/scripts/motan/motan_graph.py -l
```

It is also possible to specify matplotlib plot options for each dataset:

```
~/klipper/scripts/motan/motan_graph.py mylog -g
'[["trapq(toolhead,velocity)?color=red&alpha=0.4"]]'
```

Many matplotlib options are available; some examples are "color", "label", "alpha", and "linestyle".

The `motan_graph.py` tool supports several other command-line options - use the `--help` option to see a list. It may also be convenient to view/modify the motan_graph.py script itself.

The raw data logs produced by the `data_logger.py` tool follow the format described in the API Server. It may be useful to inspect the data with a Unix command like the following: `gunzip < mylog.json.gz | tr '\03' '\n' | less`

Generating load graphs

The Klippy log file (/tmp/klippy.log) stores statistics on bandwidth, micro-controller load, and host buffer load. It can be useful to graph these statistics after a print.

To generate a graph, a one time step is necessary to install the "matplotlib" package:

```
sudo apt-get update
sudo apt-get install python-matplotlib
```

Then graphs can be produced with:

```
~/klipper/scripts/graphstats.py /tmp/klippy.log -o loadgraph.png
```

One can then view the resulting **loadgraph.png** file.

Different graphs can be produced. For more information run: `~/klipper/scripts/graphstats.py --help`

Extracting information from the klippy.log file

The Klippy log file (/tmp/klippy.log) also contains debugging information. There is a logextract.py script that may be useful when analyzing a micro-controller shutdown or similar problem. It is typically run with something like:

```
mkdir work_directory
cd work_directory
cp /tmp/klippy.log .
~/klipper/scripts/logextract.py ./klippy.log
```

The script will extract the printer config file and will extract MCU shutdown information. The information dumps from an MCU shutdown (if present) will be reordered by timestamp to assist in diagnosing cause and effect scenarios.

Testing with simulavr

The simulavr tool enables one to simulate an Atmel ATmega micro-controller. This section describes how one can run test gcode files through simulavr. It is recommended to run this on a desktop class machine (not a Raspberry Pi) as it does require significant cpu to run efficiently.

To use simulavr, download the simulavr package and compile with python support. Note that the build system may need to have some packages (such as swig) installed in order to build the python module.

```
git clone git://git.savannah.nongnu.org/simulavr.git
cd simulavr
make python
make build
```

Make sure a file like **./build/pysimulavr/_pysimulavr.*.so** is present after the above compilation:

```
ls ./build/pysimulavr/_pysimulavr.*.so
```

This commmand should report a specific file (e.g. **./build/pysimulavr/_pysimulavr.cpython-39-x86_64-linux-gnu.so**) and not an error.

If you are on a Debian-based system (Debian, Ubuntu, etc.) you can install the following packages and generate *.deb files for system-wide installation of simulavr:

```
sudo apt update
sudo apt install g++ make cmake swig rst2pdf help2man texinfo
make cfgclean python debian
sudo dpkg -i build/debian/python3-simulavr*.deb
```

To compile Klipper for use in simulavr, run:

```
cd /path/to/klipper
make menuconfig
```

and compile the micro-controller software for an AVR atmega644p and select SIMULAVR software emulation support. Then one can compile Klipper (run `make`) and then start the simulation with:

```
PYTHONPATH=/path/to/simulavr/build/pysimulavr/ ./scripts/avrsim.py
out/klipper.elf
```

Note that if you have installed python3-simulavr system-wide, you do not need to set `PYTHONPATH`, and can simply run the simulator as

```
./scripts/avrsim.py out/klipper.elf
```

Then, with simulavr running in another window, one can run the following to read gcode from a file (eg, "test.gcode"), process it with Klippy, and send it to Klipper running in simulavr (see installation for the steps necessary to build the python virtual environment):

```
~/klippy-env/bin/python ./klippy/klippy.py config/generic-simulavr.cfg -i
test.gcode -v
```

Using simulavr with gtkwave

One useful feature of simulavr is its ability to create signal wave generation files with the exact timing of events. To do this, follow the directions above, but run avrsim.py with a command-line like the following:

```
PYTHONPATH=/path/to/simulavr/src/python/ ./scripts/avrsim.py
out/klipper.elf -t PORTA.PORT,PORTC.PORT
```

The above would create a file **avrsim.vcd** with information on each change to the GPIOs on PORTA and PORTB. This could then be viewed using gtkwave with:

```
gtkwave avrsim.vcd
```