

# ITC-QEMU-GUI v.0.2 Documentation

August 17, 2021

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
1.1	Purpose . . . . .	2
1.2	Installation . . . . .	2
<b>2</b>	<b>Features</b>	<b>3</b>
2.1	Memory Dump . . . . .	3
2.1.1	Reading Memory . . . . .	3
2.1.2	Displaying Memory . . . . .	3
2.1.3	Auto Refresh . . . . .	4
2.1.4	Searching . . . . .	4
2.1.5	Saving . . . . .	4
2.2	Memory Tree View . . . . .	4
2.3	Assembly View . . . . .	5
2.4	CPU Register View . . . . .	5
2.4.1	Searching Events . . . . .	6
2.4.2	Viewing and Saving Output . . . . .	7
2.5	Logging View . . . . .	7
2.6	Time Multiplier Graph . . . . .	8
2.7	Miscellaneous . . . . .	9
2.7.1	Dark Mode . . . . .	9
2.7.2	Plugin Support . . . . .	9
<b>3</b>	<b>Added QMP Commands</b>	<b>10</b>
3.1	get-pmem . . . . .	10
3.2	mtree . . . . .	10
3.3	itc-sim-time . . . . .	11
3.4	itc-time-metric . . . . .	11
3.5	itc-cpureg . . . . .	12
<b>4</b>	<b>Future Work as of v.0.2</b>	<b>12</b>

## Revision History

Revision	Date	Author(s)	Description
v.0.1	July 8, 2020	Ian Peitzsch, Michael Hoefler	Created
v.0.2	August 3, 2021	Rachel Misbin	Revised

# 1 Introduction

## 1.1 Purpose

The purpose of this document is to explain the installation process, as well as the use and functionality of ITC QEMU GUI v.0.2. This codebase can be publicly accessed at <https://github.com/rom81/itc-qemu-gui/tree/master>.

## 1.2 Installation

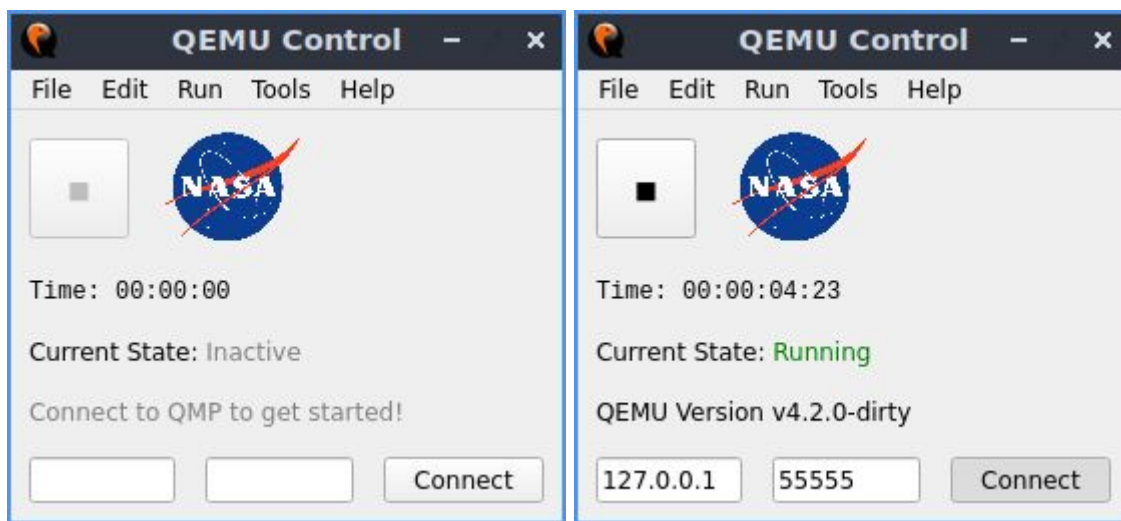
Before starting to install the GUI, make sure you have the following dependencies installed on your machine. These are all available to install from most Linux package managers.

- git
- build-essential
- python3
- python3-virtualenv

To install the GUI, you should first create a custom build of QEMU using the provided `qemu_install.sh` script. This is necessary because the GUI takes advantage of some commands which are custom and are not available in vanilla QEMU. After using the script to build QEMU, the `startup.sh` script is used to setup and run the GUI. Since the script must be run in the context of the current terminal, make sure to run it using `source startup.sh`. The script will install a python virtual environment, install necessary packages in the environment, and run the GUI. This command may be used each time you start the GUI.

To start using the GUI, you should first run an instance of QEMU, making sure to make QMP accessible. A simple example is listed below (which assumes that you have already created a bootable alpine image). If you are unsure how to get started with QEMU, refer to the `itc-qemu-manual`.

```
$ qemu-system-x86_64 -qmp tcp:127.0.0.1:55555,server,nowait ../alpine/alpine.img
```

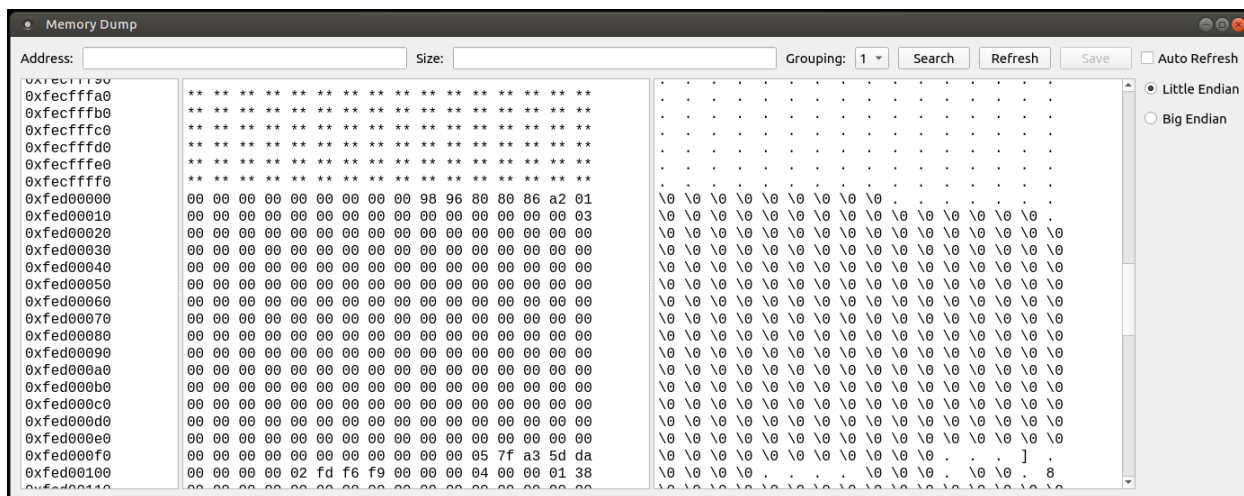


Once you complete the steps in the previous section you should see the main control panel of the GUI pictured above (left). If you are running QMP on port 55555 as in the example, you can just press the connect button without typing anything, and the GUI will automatically connect you to `localhost` on port 55555 as seen in the image on the right. Otherwise, you should type in the QMP host and port, and click the connect button. From top to bottom, you can see the pause / play button, the simulation time, the simulation state, the QEMU version, and the connect dialog. Next, we can look at some of the more advanced features that the GUI offers.

## 2 Features

The GUI offers many features that aid with debugging and using QEMU in general. The simplest feature is arguably the pause / play button on the main screen, which uses QMP to send "stop" and "continue" signals to the simulation. The following section will explore some of the more advanced features of the GUI such as the CPU Register View and the Memory dump. Keep in mind that some features of the GUI make use of not only QMP commands, but also HMP commands. To invoke HMP commands through QMP, the GUI uses the following syntax `{"execute": "human-monitor-command", "arguments": {"command-line": [HMP_COMMAND]}}`

### 2.1 Memory Dump



The memory dump view allows users to view the contents of memory. This view displays the memory address, a hexadecimal representation of memory, and an ASCII representation of memory. To open the memory dump view, navigate to Tools → Memory Dump.

#### 2.1.1 Reading Memory

To read a part of memory, specify the base address in the **Address** field. Additionally, specify the number of bytes to read in the **Size** field. If **Size** exceeds 2048, then it will be set to 2048. Finally, click **Refresh** to fetch **Size** bytes of memory starting **Address**.

It is also possible to scroll up or down to load adjacent regions of memory.

#### 2.1.2 Displaying Memory

This view automatically displays memory as bytes represented as hexadecimal and ASCII values. The **Grouping** dropdown menu can be used to set the grouping of bytes in the hexadecimal display. **Grouping** can be either 1, 2, 4, or 8, corresponding to grouping bytes, 2 bytes, 4 bytes, and 8 bytes.

On the right-hand side of the display, there are 2 possible selections for endianness: little endian and big endian. Selecting these displays the memory with the selected endianness. The default endianness is little endian.

For either of these display changes to take effect, the display must be refreshed which can be accomplished either by clicking **Refresh** or by having **Auto Refresh** selected.

### 2.1.3 Auto Refresh

The **Auto Refresh** checkbox in the top right corner controls whether the display auto refreshes or not. With **Auto Refresh** selected, the display will refresh at a rate of 10Hz. By default, **Auto Refresh** is checked.

**Note:** The auto refresh only refreshes the currently displayed region of memory. To load a different region the user must either scroll or follow directions as specified in [2.1.2](#)

### 2.1.4 Searching

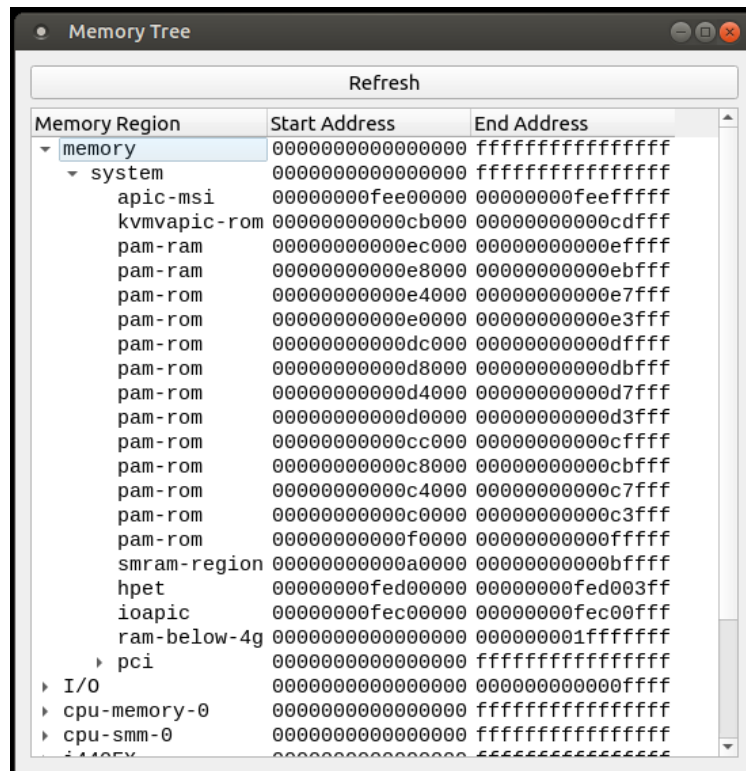
The **Search** button lets users search for memory at a specified address. If the address is within the currently displayed range, then the address, hexadecimal value at that address, and the ASCII value at that address will be highlighted, and the view will scroll to make them visible. If the address is not within the currently displayed range, then 1024 bytes starting at the specified address will be loaded in, and the same values will be highlighted.

### 2.1.5 Saving

The **Save** button allows the user to save the currently displayed range of memory to a file. To enable the **Save** button, the simulation must be paused.

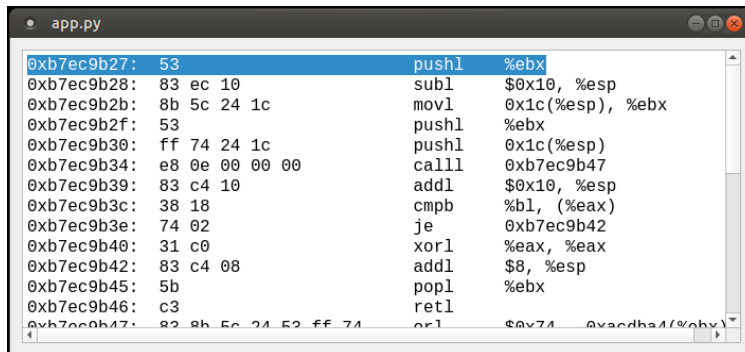
**Note:** **Save** uses `pmemsave` to save memory to the file, so any grouping or endian display changes will not be present in the file.

## 2.2 Memory Tree View



The memory tree view displays the different regions of memory and their address ranges in the form of a tree. To open the memory tree view navigate to Tools → Memory Tree. Each region can be expanded to show its subregions. Additionally, double clicking a region opens up a memory dump view with that region displaying.

## 2.3 Assembly View



```
0xb7ec9b27: 53                pushl   %ebx
0xb7ec9b28: 83 ec 10          subl   $0x10, %esp
0xb7ec9b2b: 8b 5c 24 1c       movl   0x1c(%esp), %ebx
0xb7ec9b2f: 53                pushl   %ebx
0xb7ec9b30: ff 74 24 1c       pushl   0x1c(%esp)
0xb7ec9b34: e8 0e 00 00 00    calll  0xb7ec9b47
0xb7ec9b39: 83 c4 10          addl   $0x10, %esp
0xb7ec9b3c: 38 18             cmpb   %bl, (%eax)
0xb7ec9b3e: 74 02             je     0xb7ec9b42
0xb7ec9b40: 31 c0             xorl   %eax, %eax
0xb7ec9b42: 83 c4 08          addl   $8, %esp
0xb7ec9b45: 5b               popl   %ebx
0xb7ec9b46: c3               retl
0xb7ec9b47: 82 8b 5c 24 1c  ff 74  0x74  0x1c(%esp)
```

The assembly view shows the current assembly instruction. To open the assembly view navigate to Tools → Assembly View. The display updates when the GUI receives a stop signal from QEMU, allowing it to work in tandem with GDB using the `si` or `ni` commands.

**Note:** To properly use this view, the architecture's instruction pointer register must be set in `package/constants.py`. By default, `$eip` is used.

### Areas for Improvement:

- A previous version of tool explored using QEMU's existing gdbstub functionality to perform be able to step through assembly, rather than just using an HMP command (as is done in the current version of this software). The current version of this software does not include this functionality because it did not work reliably, however, if this functionality is desired (despite its flaws), an experimental version is located under the branch `gdb-fixes-in-progress`. In short, the bug preventing this functionality from existing in the main version of this GUI is because the `si` and `ni` do not always give a stop signal when they finish stepping. Thus, sometimes the display doesn't properly update. This functionality is currently commented out in the main version.

## 2.4 CPU Register View

The CPU register view is used to view simulation CPU registers and their contents. The view can be opened by navigating to (Tools → CPU Register View).

CPU Registers			
File Options			
RAX	ffffffff976e6980	RBX	0000000000000000
RDX	0000000000002e96	RSI	0000000000000087
RBP	0000000000000000	RSP	ffffffff98003ea0
R9	0000000000000006	R10	0000000000000000
R12	ffffffff98013780	R13	0000000000000000
R15	ffffffff98013780	RIP	ffffffff976e6d5e
CPL	0	II	0
SMM	0	HLT	1
CS	ffffffff 00af9b00	DPL	0
SS	ffffffff 00cf9300	DS	00000000 00000000
GS	00000000 00000000	LDT	
TS564-avl		GDT	00001000 0000007f
CR0	80050033	CR2	00007f1643632d13
CR4	000006f0	DR0	0000000000000000
DR2	0000000000000000	DR3	0000000000000000
DR7	0000000000000400	EFER	000000000000d01
FSW	0000 [ST 0]	FTW	00
FPR0	000000000000 0000	FPR1	000000000000 0000
FPR3	000000000000 0000	FPR4	000000000000 0000
FPR6	000000000000 0000	FPR7	000000000000 0000
XMM01	0000000000000000	XMM02	0000000000000000
XMM04	0000000000000000	XMM05	0000000000000000
XMM07	0000000000000000	XMM08	0000000000000000
XMM10	0000000000000000	XMM11	0000000000000000
XMM13	0000000000000000	XMM14	0000000000000000
		XMM15	0000000000000000
		RCX	0000000000000001
		RDI	0000000000000087
		R8	ffff8a4246e1db40
		R11	000000000000002a
		R14	0000000000000000
		RFL	0000246 [---Z-P-]
		A20	1
		ES	00000000 00000000
		CS64	[-RA]
		FS	00000000 00000000
		TR	0000206f 00008900
		IDT	00000000 00000fff
		CR3	00000000015f2000
		DR1	0000000000000000
		DR6	00000000ffff0fff0
		FCW	037f
		MXCSR	00001f80
		FPR2	000000000000 0000
		FPR5	000000000000 0000
		XMM00	00000003000000010
		XMM03	0000000000000000
		XMM06	0000000000000000
		XMM09	0000000000000000
		XMM12	0000000000000000

The register view offers options to toggle auto-refresh and to save the registers to a separate file. Finally, text mode can be toggled through the Options menu, and offers a less visually pleasing (but more organized and compact) view of the registers.

### 2.4.1 Searching Events

In the top left of the trace events screen you should see a search bar. You can use this search bar to search for trace events. To the right of the search bar you should see an up arrow and a down arrow. The up arrow is used to collapse all nodes in the tree of trace events, and the down arrow is used to expand all of the nodes.

### 2.4.2 Viewing and Saving Output

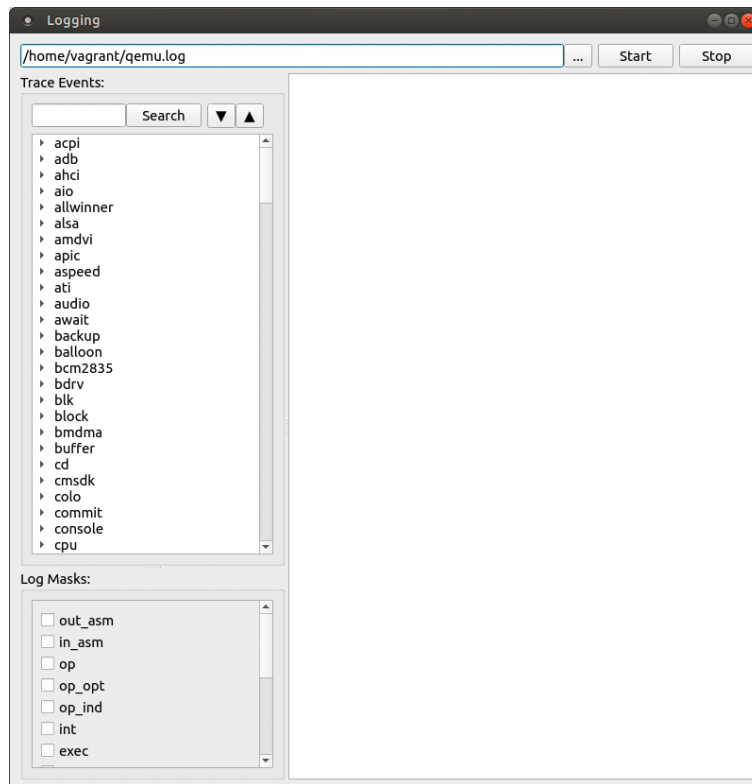
To the right of the trace event listing you should see the past one hundred trace events. This output is saved in a file located at `/tmp/errors.log` and regex is used to filter out normal logging output (leaving only trace events). To save the output that you can see on the right of the screen to a file, navigate to File → Save to File. To disable auto-refresh, you may navigate to Options → Auto Refresh.

#### Note:

- The trace event window actually supports custom trace events which makes it very useful for debugging code that you may have written to customize and add to QEMU.

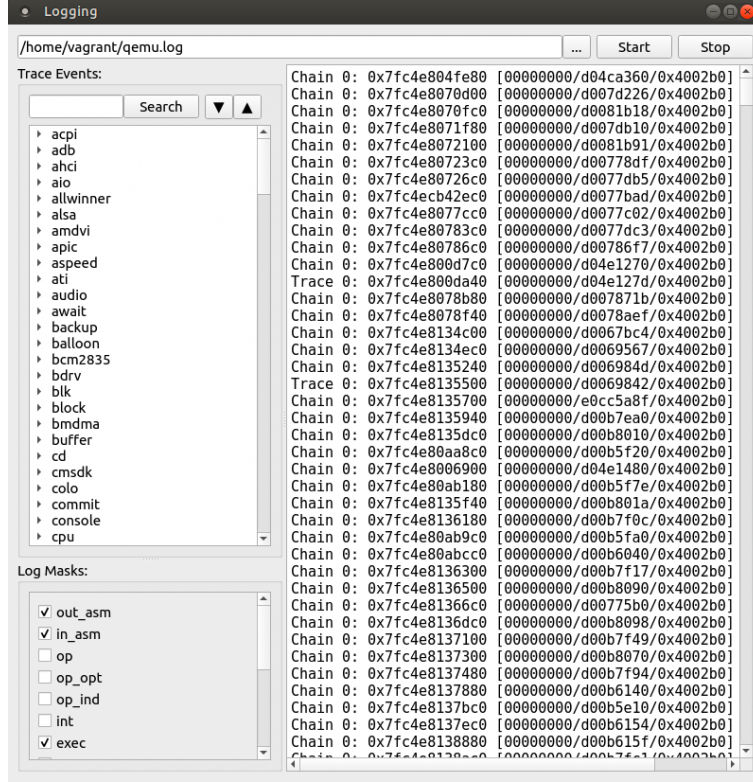
## 2.5 Logging View

The logging view allows you to view, save, and filter trace event logs. To open the log navigate to Tools → Logging. Whenever you select a logging option, output is stored to a logfile located at the default logging location (`qemu.log` in the user's home directory) or at the path specified in file prompt. The view can be seen below.

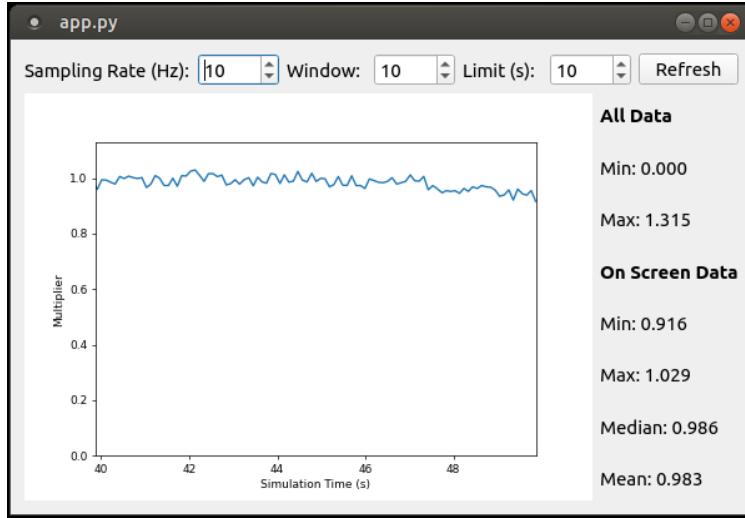


By default, the display will auto-refresh. There are two ways to select events to log: (1) by selecting a specific trace event (in the left dropdown view) or (2) by selecting a log mask (under the trace event dropdown view). The text entry box with the **Search** button can be used to search through the trace events dropdown view. The **Start** and **Stop** buttons can be used to start and stop logging. When logging is started, the log view is also displayed as shown below.





## 2.6 Time Multiplier Graph



The time multiplier graphs real time vs.  $\frac{\Delta t_{sim}}{\Delta t_{real}}$  where  $t_{sim}$  is the simulation's virtual time, and  $t_{real}$  is the real time. To open the time multiplier graph navigate to Tools → Time Multiplier.

At the top of the display, there are 3 values to alter the graph: **Sampling Rate**, **Window**, and **Limit**. **Sampling Rate**, as its name suggests, is the rate at which the simulation's time is sampled. **Window** is the number of samples used for the moving average. So, a smaller value for **Window** makes the graph change quicker, while a larger value makes the graph change slower. **Limit** is the number of seconds to display. After changing any of these values, click **Refresh** to make the changes take effect. The default value for all 3 of these values is 10.

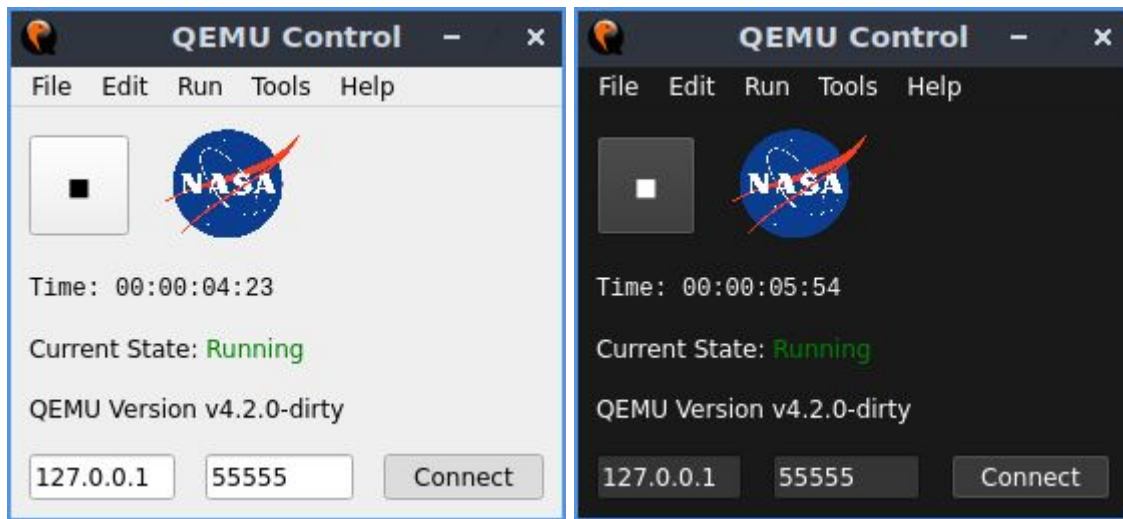


On the right-hand side there are various data about the graph, including the absolute minimum, absolute maximum, and the minimum, maximum, mean, and median for values currently displayed.

## 2.7 Miscellaneous

### 2.7.1 Dark Mode

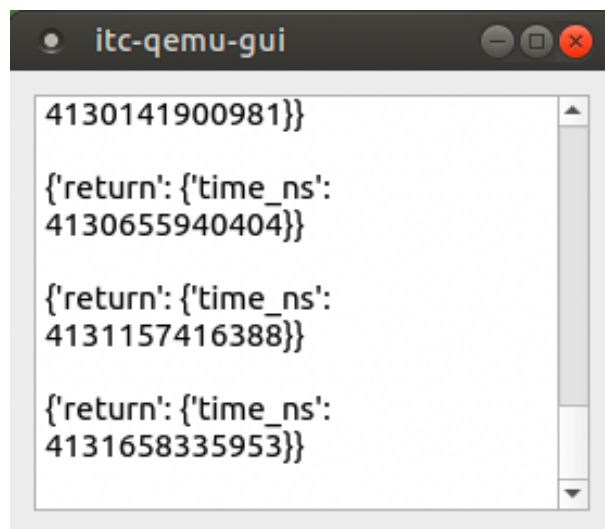
Dark mode can be enabled through the preferences dialog which contains the checkbox used to toggle dark mode (Edit → Preferences).



### 2.7.2 Plugin Support

This system uses [Yapsy](#) as a plugin manager. To add a plugin after implementing it, specify the plugin path in `addPlugins` in `package/mainwindow.py`. Then also specify how the plugin should be handled.

All plugins should be accessible through the Plugins dropdown menu, which can be found by doing Tools → Plugins. Right now, the only plugin is the QMP Display which shows responses to QMP commands, as shown below.



## 3 Added QMP Commands

### 3.1 get-pmem

The `get-pmem` command requests the contents of a part of physical memory.

#### Arguments

- `addr: int64`  
The starting address of the requested part of memory.
- `size: int64`  
The number of bytes of memory being requested.
- `hash: int64`  
A unique identifier.

**Return** `get-pmem` returns a `MemReturn` object

#### `MemReturn`

- `hash: int64`  
The unique identifier passed to `get-pmem`.
- `vals: MemVal[]`  
An array of `MemVal` objects

#### `MemVal`

- `val: uint64`  
Single value of memory. The number of bytes `val` represents depends on the value of `grouping`.
- `ismapped: bool`  
Indicates whether that value is from a mapped region of memory or not.

#### Example:

```
-> {"execute": "get-pmem",
  "arguments": {
    "addr": 4275044352,
    "size": 1024,
    "hash": 0,
    "grouping": 1 } }
<- {"return": {"hash": 0, "vals": [{"val": 0, "ismapped": true}, {"val": 0, "ismapped":
true}, ... ] }
```

### 3.2 mtree

The `mtree` command returns a tree representation of memory regions.

**Arguments** `mtree` takes no arguments.

**Return** `mtree` returns an array of `MemoryMapEntry` objects. This array is a depth-first traversal of the tree.

## MemoryMapEntry

- **name:** str  
The name of the region of memory.
- **start:** int  
The start address of the region of memory.
- **end:** int  
The end address of the region of memory.
- **parent:** str  
The name of the parent region. A value of "" indicates it is a root node. There can be multiple roots.

### Example:

```
-> {"execute": "mtree"}
<- {"return": { {"name": "memory", "start": 0, "end": -1, "parent": ""}, {"name": "system", "start": 0, "end": -1, "parent": "memory"}, ... } }
```

## 3.3 itc-sim-time

The `itc-sim-time` command returns the time in nanoseconds given a specified `ClockType`. `ClockType` is an enum with possible values `realtime`, `virtual`, `host`, and `virtual-rt` which all correspond to QEMU's clock type enum in `include/qemu/timer.h`.

### Arguments

- **clock:** ClockType  
The clock type to use to get the time.

**Return** `itc-sim-time` returns a `SimTime` object.

## SimTime

- **time\_ns:** int64  
The time in nanoseconds.

### Example:

```
->{"execute": "itc-sim-time",
"arguments": {"clock": "virtual"} }
<-{"return": {"time_ns": 1040040609 } }
```

## 3.4 itc-time-metric

The `itc-time-metric` command gives an array containing the current host time and the current virtual time.

**Arguments** `itc-time-metric` does not take any arguments.

**Return** `itc-time-metric` returns an array containing 2 `SimTime` objects. The first value is the virtual time, and the second value is the host time.

### Example:

```
->{"execute": "itc-time-metric" }
<-{"return": { {"time_ns": 1040040609}, {"time_ns": 1594065163483704} } }
```

### 3.5 itc-cpureg

The `itc-cpureg` command returns all CPU register names and values along with the total CPU register count. In v.0.1 of the GUI, the CPU Register View used the HMP command `info registers` but this necessitated complex parsing in the GUI (since HMP commands are returned as strings, rather than as JSON objects as in QMP commands) which was very fragile and broke with QEMU updates. It was suggested that a custom QMP command might eliminate this issue, so the `itc-cpureg` command was developed. Since the `info registers` command used a function which dumped the CPU state to a file, it was copied then modified to create a function to return the CPU state as a JSON object rather than writing it to a file.

It is important to understand that the `itc-cpureg` command is currently only implemented for i386 systems; since CPU registers vary across architectures, there is no one-size-fits-all solution for all architectures. To extend this command to work for other architectures, a function `[ARCH]_cpu_return_state` needs to be written and connected to the `CPUClass` function `return_state`. This function will look very similar to that architecture's `[ARCH]_cpu_dump_state` function, except it will return the CPU state instead of writing it to a file. Currently, `x86_cpu_return_state` is the only example to follow in writing this function. Once this function is written and connected to `CPUClass`'s `return_state` function, the `itc-cpureg` command will work for that architecture.

**Arguments** `itc-cpureg` does not take any arguments.

**Return** `itc-cpureg` returns an array containing a `CpuReturn` object and an integer representing the total number of registers described in the array.

## 4 Future Work as of v.0.2

In inspecting an upgrading this software from v.0.1 to v.0.2, several bug fixes and improvements were identified and made, however, some work remains to further improve this application. This includes:

- The version of the Assembly View which uses GDB (described in the section on the Assembly View) could be debugged to enable stepping.
- In the QMP Wrapper (`qmpwrapper.py`), socket communications with QEMU could be made more robust by replacing the check for termination characters at the end of received messages with a full-featured messaging framework to ensure complete and reliable reception of packets from QEMU.
- A known race condition around accessing the window's scroll bar exists in the Memory Dump View. Its impact has been reduced by tightening the window of access to this variable, therefore tightening the critical section but this race condition could still theoretically occur. A more thorough investigation of the state of this race condition and potentially a refactor of `memdump.py` would solve this.